العنوان: Software Product Line Engineering Based on Web Services
المؤلف الرئيسي: Saleh, Mazen M. Aquil
مؤلفين آخرين: Gomaa, Hassan(Super.)
التاريخ الميلادي: 2005
موقع: فيرفاكس، فرجينيا
رقم MD: 618453
نوع المحتوى: رسائل جامعية
اللغة: English
الدرجة العلمية: رسالة دكتوراه
الجامعة: George Mason University
الكلية: Volgenau School of Engineering
الدولة: الولايات المتحدة الأمريكية
قواعد المعلومات: Dissertations
مواضيع: البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط: https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

1. INTRODUCTION

1.1 Background

The field of software reuse has evolved from reuse of individual components towards large-scale reuse with software product lines [Clements02]. A software product line (SPL) consists of a family of software systems that have some common functionality and some variable functionality. Parnas referred to a collection of systems that share common characteristics as a *family of systems* [Parnas79]. According to Parnas, it is worth considering the development of a family of systems when there is more to be gained by analyzing the systems collectively rather than separately, i.e. the systems have more features in common than features that distinguish them. A *family of systems* is now referred to as a *software product line* or *software product family*.

A Software Product Line (SPL) is developed by engineering a reusable architecture for the product line, which can be configured to generate target applications [Gomaa99, Gomaa04]. The two major activities used in developing product lines are SPL engineering and application engineering. SPL engineering involves the analysis, design, and implementation of product line software that satisfy the requirements of the families of systems [Weiss99, Gomaa04]. Application engineering involves tailoring the

engineered SPL to produce target applications based on a given set of configuration requirements [Sugumaran92, Gomaa04].

This dissertation addresses product lines based on web services. A web service is defined as a collection of functional methods that are grouped into a single package and published in the Internet for use by other applications. Web services use the standard Extensible Markup Language (XML) to exchange information with other software via the Internet protocols [Deitel et al. 2003, Howard04, Booth04].

Although there is much research into software product line engineering, this research extends product line concepts to address the engineering and customization of product lines that are based on web services.

1.2 Research Problem and Approach

This research focuses on designing, developing and customizing software product lines based on web services to derive executable target applications from the product line using an automated customization environment. The approach taken is to:

a) Develop a design approach for software product line service-oriented architecture.
b) Introduce three different development approaches to support the automatic customization of SPL architecture and components:
c) Develop a proof-of-concept prototype to support this research

d) Validate this research with two web services-based software product line case studies.

1.3 Importance and Rationale of This Research

The idea of web services has been strongly promoted in industry by companies such as Microsoft, IBM, Oracle, and Hewlett-Packard. They see this new technology as a broad new vision for how software systems are analyzed, developed, and used [McDougall 01]. Web services employ open standards that are text-based, which introduce a new approach to communication between heterogeneous platforms and applications [Deitel 03]. Using the already existing internet technology, web services make communication, interoperability, and integration cheaper and easier to achieve, compared to current methods, such as CORBA and DCOM [Deitel 03]. As the use of web services continues to grow, software product lines engineers should take full advantage of this technology. Therefore, it is essential to develop a new methodology that enables the design, development, and customization of software product lines that consist of web services-based components.

1.4 Terminology

This section provides definitions of important terms used in this dissertation.

Unified Modeling Language

Unified Modeling Language (UML) is a standardized object-oriented development environment that is used to analyze and design systems.

Software Product Line

A software product line (SPL) is a family of systems that share common features. It is developed by engineering an application domain that can be configured to generate target systems through the customization process of selecting optional and alternative features. [Parnas79, Gomaa04]

Feature

A feature is a functional requirement of a software application.

SPL Engineer

The SPL engineer is responsible for designing and developing the product line.

Application Engineer

The application engineer is responsible for customizing the product line to derive target applications.

Kernel Source Code

Kernel source code refers to source code that exists in all derived target applications.

Variable Source Code

Variable source code refers to optional or alternative source code blocks that are integrated with kernel source code based on feature selection to produce a customized target application.

Separation of Concerns

Separation of concerns refers to the separation of common and variable product line concerns. It involves the separation of variable source code from kernel source code into a variable source code file.

Code Weaving

Code weaving is the integration of kernel source code with optional and alternative source code

Client application

Client application refers to the client subsystem and the software objects it contains.

Server application

Server application refers to the server subsystem and its constituent web service components and database.

1.5 Organization

The rest of the dissertation is organized as follows. Chapter 2 contains an overview of related work. Chapter 3 addresses the problem statement and research approach, including comparison of related work with this research effort. Chapter 4 describes the proposed design approach using a Hotel System case study. Chapter 5 describes the three development approaches and their customization environment. Chapter 6 describes the proof-of-concept prototype that is used to support this research. Chapter 7 includes contributions and future research. References and appendices are attached at the end, including the second case study of Radio Frequency Management System.

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

2. RELATED WORK

2.1 Introduction

This chapter surveys other research efforts that are related to the research described in this dissertation. This chapter begins by defining software product lines in section 2.2. Section 2.3 describes the Evolutionary Software Product Line Engineering Process (PLUS). Section 2.4 describes the multiple-view model of software product lines used in the PLUS environment. Section 2.5 addresses other software product line engineering methods. 2.6 describes component-based software engineering. Web services are described in section 2.7. Section 2.8 describes Aspect-Oriented Programming, and section 2.9 describes frame technology.

2.2 Software Product Lines

A software product line is a family of systems that share common features [Gomaa92, Gomaa04]. It is developed by engineering a Software Product Line (SPL) that can be tailored to generate target systems [Gomaa99, Farrukh98, Weiss99]. Software product line engineering involves the analysis, design, and implementation of a product line that satisfies the requirements of all target applications [Sugumaran92, Gomaa04]. This can be achieved by capturing the commonality and variability of a family of system at the analysis phase, and applying this information at the design and implementation phases

[Gomaa 99]. "The goal of software product families is to improve productivity through software reuse. A new application system can be configured from the domain model given the common features (requirement) of the domain and variable features that reflect differences among the members of the product family" [Farrukh 1998].

2.3 Evolutionary Software Product Line Engineering Process

The Evolutionary Software Product Line Engineering Process (PLUS) [Gomaa04] consists of two main processes, as shown in Figure 2-1:

a) Software Product line Engineering. A product line multiple-view model, which addresses the multiple views of a software product line, is developed. The product line multiple-view model, product line architecture, and reusable components are developed and stored in the product line reuse library.

b) Application engineering. Involves the configuration of target applications from the SPL architecture and implementation. A target application is a member of the software product line. The multiple-view model for a target application is configured from the product line multiple-view model. The user selects the desired features for the product line member (referred to as target application). Given the target application features, the product line model and architecture are adapted and tailored to derive the target application model and architecture. The architecture determines which of the reusable components are needed for configuring the executable target application.

Earlier papers have described how this approach was carried out before [Gomaa96, Gomaa99] and after the introduction of the UML [Gomaa02, Gomaa04]. This research describes how product line engineering can be carried out for product lines that are based on Web Services.

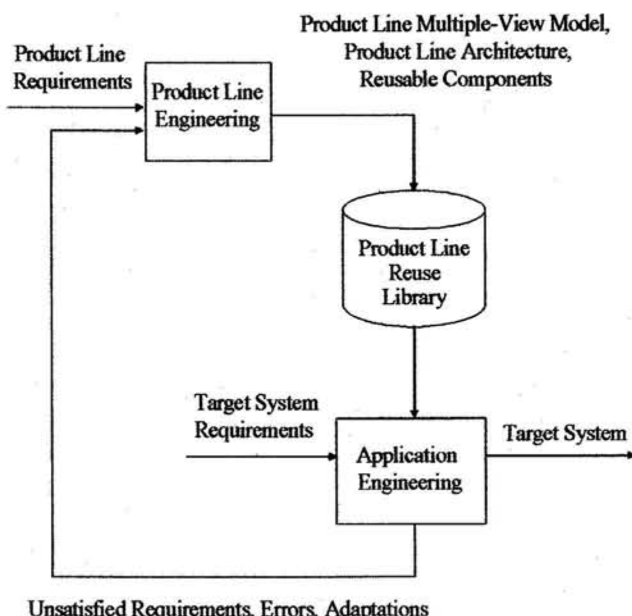


Figure 2-1 Evolutionary Software Product Line Engineering Process

2.4 Multiple-View Models of Software Product Lines

A multiple-view model for a software product line defines the different characteristics of a software family [Parnas79], including the commonality and variability among the members of the family [Clements02, Weiss99]. A multiple-view model is represented using the UML notation [Rumbaugh99, Gomaa00a, Gomaa04] and considers the product

line from different perspectives. The PLUS environment [Gomaa04] is based on the multiple-view mode for software product lines, as described in the following sections.

2.4.1 Use Case Model for Software Product Lines

The functional requirements of a system are defined in terms of use cases and actors [Jacobson97]. An actor is a user type. A use case describes the sequence of interactions between the actor and the system, considered as a black box.

For a single system, all use cases are required. When modeling a software product line, kernel use cases are those use cases required by all members of the family. Optional use cases are those use cases required by some but not all members of the family. Some use cases may be alternative, that is different versions of the use case are required by different members of the family [Gomaa04].

2.4.2. Feature Analysis for Software Product Lines

Feature analysis is an important aspect of domain analysis [Cohen98, Gomaa04, Griss98, Kang90]. In domain analysis, features are analyzed and categorized as kernel features (must be supported in all target systems), optional features (only required in some target systems), and prerequisite features (dependent upon other features). There may also be dependencies among features, such as mutually exclusive features. The emphasis in feature analysis is on the optional and alternative features, since they differentiate one member of the family from the others. In modeling software product lines, features may be functional features (addressing software functional requirements), non-functional

features (e.g., relating to security or performance), or parametric features (e.g., parameter whose value can be set differently in different members of the product line).

In the object-oriented analysis of single systems, use cases are used to determine the functional features of a system. They can also serve this purpose in product families. Griss [Griss98] has pointed out that the goal of the use case analysis is to get a good understanding of the functional requirements whereas the goal of feature analysis is to enable reuse. Use cases and features may be used to complement each other. In particular, use cases can be mapped to features based on their reuse properties.

Functional requirements that are required by all members of the family are packaged into a kernel feature. From a use case perspective, this means that the kernel use cases, which are required by all members of the family, constitute the kernel feature. Optional use cases, which are always used together, may also be packaged into an optional feature.

2.4.3 Static Model for Software Product Lines

A static model for a product line has kernel classes, which are used by all members of the product family, and optional classes that are used by some but not all members of the family. Variants of a class, which are used by different members of the product family, can be modeled using a generalization / specialization hierarchy. UML stereotypes are used to allow new modeling elements, tailored to the modeler's problem, which are based on existing modeling elements [Booch99, Rumbaugh99]. Thus, the stereotypes

<<kernel>>, <<optional>>, and <<variant>> are used to distinguish between kernel, optional, and variant classes [Gomaa04].

2.4.4 Collaboration Model for Software Product Lines

The collaboration model is used to depict the objects that participate in each use case, and the sequence of messages passed between them [Rumbaugh99, Gomaa00]. In product families, the collaboration model is developed for each use case, kernel or optional. Once the use cases have been determined and categorized as kernel, optional, or variant, the collaboration diagrams can be developed [Gomaa04].

For each feature, the objects that are needed to support the feature are determined and depicted on a feature based collaboration diagram. With this UML based approach, the objects are determined from the use cases. It should be noted that on the feature based collaboration diagram, the message sequence numbering, which is shown on individual use case based collaborations, is usually not shown.

This view is very important as it is used to determine how the objects interact with each other to support a given use case. The interconnected objects in a collaboration diagram supporting one use case depend on (and hence communicate with) objects supporting a prerequisite use case.

2.5 Other Software Product Line Engineering Methods

There are several domain engineering methods that are used to develop family of systems, such as FODA [Kang90, Cohen98], RSEB [Jacobson92, Jacobson97], FAST [Weiss99], and KobrA [Atkinson00]. The above product line engineering methods are described in the following sections.

2.5.1 Feature-Oriented Domain Analysis (FODA)

Feature-Oriented Domain Analysis (FODA) is a domain analysis method that is used to define a family of systems [Kang90, Cohen98]. It consists of:

a) Context analysis: Analyzes the scope of a domain. In this phase, a context model for the product line is developed using context diagrams. In this analysis, relationships between the product line and external elements are analyzed, and the variability is identified.

b) Domain modeling: Identifies commonalities and differences in a family of systems. Multiple models are developed to represent the specified product line. The feature model is the heart of the FODA method. It represents the relationships among the features as a hierarchical tree. Some other models used here are entity-relationship models and functional models.

c) Architecture modeling: Models a generic software architecture for a family of systems using the product line models. It defines the process for allocating the features, functions, and data objects defined in the product line models [Kang90].

2.5.2 Reuse-driven Software Engineering Method (RSEB)

Reuse-driven Software Engineering Method (RSEB) is a use case object-oriented method that is used to develop a family of related systems, where variability is modeled in the use cases using variation points that use the "extend" and "include" relations. Variability in use cases is introduced at these variation points [Jacobson92, Jacobson97].

The RSEB method includes several engineering processes:

a) Object-Oriented business engineering: Captures business processes based on object-oriented use cases
b) Map Business processes to information systems: This engineering process requires an analysis of the overall business, which include 3 sub-processes:
   - Application family engineering, which involves the development of a domain model and a domain architecture
   - Component system engineering, which involves the development of components based on the domain model and architecture
   - Application system engineering, which involves the development of new application systems using application family architecture and components [Jacabson92, Jacabson97]

2.5.3 FAST

Weiss and Lai have proposed the FAST method [Weiss99, Coplen98], which "incorporate abstraction and parameterization techniques into a configuration language

for modeling each family member. The configuration of each family member is mapped to templates through a source code generator" [Shin02] that is used to produce executable source code. Templates represent the variations of the family members. Variations are identified by parameter values, which are used in the configuration language when generating source code. The FAST method is used in a domain that is fully described with parameters and templates [Weiss99, Shin 2002].

2.5.4 KobrA

The KobrA method [Atkinson00] is a component-based product line development method containing two major processes:

a) Framework engineering: The process of developing a framework that is used as a reusable infrastructure for developing target systems within an application domain. This process consists of three sub processes, which are:
    - Context Realization: Determines the scope of the framework
    - Component specification: Defines requirements
    - Component realization: Designs software architecture

b) Application engineering: involves the development of target systems based on the developed framework [Atkinson00].

2.5.5 Knowledge-Based Requirement Elicitation Tool (KBRET)

The Knowledge-Based Requirement Elicitation Tool (KBRET) was developed by George Mason University for the purpose of automating the process of generating target system

specifications from a domain model [Gomaa92, Gomaa96a]. The major components of KBRET are:

a) The domain-dependent knowledge base: Derived from the object repository through a management user interface. It contains domain-specific information about a particular application [Gomaa96a].

b) The domain-independent knowledge base: "Contains the procedural and control knowledge required to generate target system specification from a domain model" [Gomaa96a].

c) The user interface manager: used to extract target system specifications based on user selection to desired features.

2.5.6 Web-Based Software Product Lines

This research was performed by Mark Gianturco [Gianturco04] to describe a new method for modeling web-based software product lines and generating target applications from them. In his research, several web-based patterns were developed to support variability in web page design and implementation. The patterns described in this research are:

- WebDesign design pattern: Describes a consistent look and feel for all web pages, by creating an object that adds all the view functionality, allowing each web page to contain the additional unique functionality.
- WebFeedback design pattern: Defines a set of objects that are used always to build a form submission page. These objects provide variable information to the submission page using stored data in the reusable entity objects.

- WebText design pattern: Defines a set of objects that are always used to build a text display page. These objects provide variable information to text pages using stored data in the reusable entity objects.
- WebLinks design pattern: Defines a set of objects that are always used to build a links page. These objects provide variable information to links page using stored data in the reusable entity objects.

The above patterns were used to customize target applications by changing the contents of the reusable entity objects to satisfy the requirement of a target application.

2.6 Component-Based Software Engineering

Component-based software engineering is concerned with the assembly of software systems from prebuilt software components where components and frameworks have to satisfy certain specifications and middleware [Bachman00]. A software component can be viewed as an architectural abstraction or as an implementation. Implementation components can be assembled and deployed into a larger system. Architectural components [Shaw96], on the other hand, express design rules in the form of a component model that imposes a set of standards to which components must conform [Kirtland99, Bachmann00].

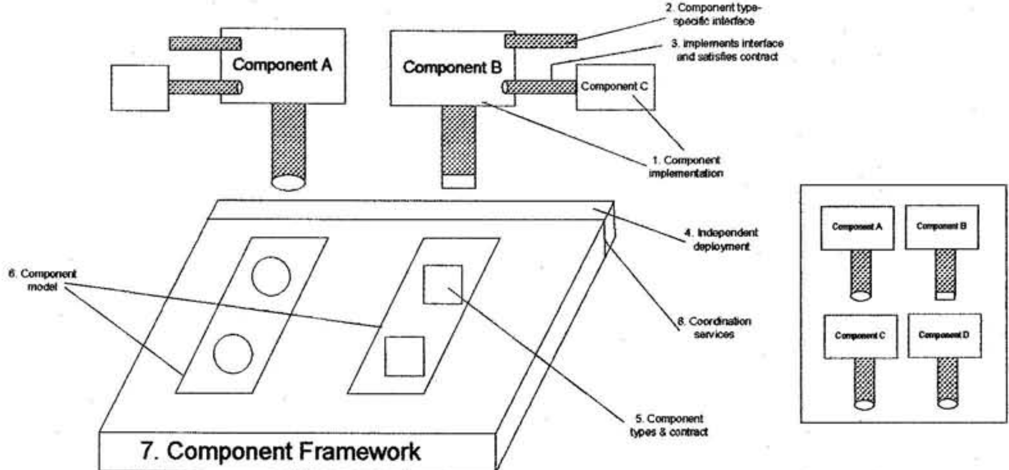


Figure 2-2 Component-Based Design Pattern [Bachmann00]

Figure 2.2 depicts an overall approach for assembling a system from prebuilt components:

1. A component - which is a software implementation of functionality;
2. Component type- specific interfaces;
3. A contract that must be met to satisfy certain tasks;
4. Independently developed components that conform to certain rules in order to interact with other components;
5. Distinct component types and contract to allow components to be assembled in a component framework;
6. A component model, which is a set of component types, interfaces, and contracts;
7. A component framework provides different runtime services to allow for component interaction;

8. Coordination services, which are runtime services that are provided by the component framework [Bachmann00, Bass00].

Component-based software applications use standards such as OMG's Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM), Sun microsystem's Remote Method Invocation (RMI), and IBM's Distributed System Object Model (DCOM) to enable communication between distributed components and to integrate different applications together [Chung03, Deitel03].

The ideas behind component-based software engineering provide great benefits to software developers, such as software reuse, improved programmer productivity, and reduce time to market. Unfortunately, this engineering software architecture has many drawbacks, which led to the new invention of "web services", discussed in the next section. Some of these drawbacks are [Bass00, Deitel03]:

- Lack of mechanism to make components interoperable: among the different standards mentioned above, interoperability is limited between them. For example, DCOM and CORBA components usually communicate via a COM/CORBA bridge. If either DCOM's or CORBA's underlying protocols change, programmers have to go through serious modifications to reflect the change.
- Each organization is using its own standards: Each organization is developing their own components based on a preferred standard to provide communication

between their developed components. As mentioned earlier, there are many standards or technologies that provide different solutions to component-based software applications.

- Platform dependent [Bass00, Deitel03].

2.7 Web Services

Web services are loosely coupled software components that use XML to exchange information with other applications over the Internet, Intranet, or Extranet [Govatos02, Deitel03, Booth04]. "The primary objective of web services is to simplify and standardize application interoperability within and across companies, leading to increased operational efficiencies and tighter partner relationships" [Govatos02].

A web service architecture consists of three primary functions [Deitel03, Howard04, Booth04]:

- Discovery: Web services are discovered through Universal Description, Discovery and Integration (UDDI). It is "a specification that defines registries in which businesses can publish information about themselves and the services they provide" [Deitel03].
- Description: Web services are described by Web Services Description Language (WSDL). It is a language that meant to be read by software applications. A WSDL document defines the messages types that a web service may send or receive. It also specifies the data that a requesting

application must provide in order for this web service to perform a specific task.

- Transport: Web services are transported using Simple Object Access Protocol (SOAP) [Deitel03, Howard04, Booth04].

2.7.1 Advantages of Web Services

Web services technology has solved many problems of its predecessors. Some of its advantages are [Deitel03]:

- Employ open standards using open, text-based, standards. XML is the main standard used for communication between web services and other web services or software applications.
- Platform independent.
- Web services are less expensive and easier to implement compared to some of the leading technologies, such as DCOM and CORBA. Web services use the available Internet protocol for their communication. Therefore, expensive private networks could be avoided. Also, since web services communicate directly without the need for a broker or any middleware, development is much simpler.
- Promote a modular approach to programming.
- Can be implemented incrementally [Deitel03].

2.7.2 Disadvantages of Web Services

Even though web services provide many benefits, they also create some challenges for application developers, such as [Deitel03]:

- Lack of standard security procedures. Web services allow direct access to a company's information resources and applications, which can expose the network to hackers and viruses. The SOAP standard protocol used in the communication process with web services does not provide security protection.
- Quality of service is one of the major challenges of web services. Response time, handling large number of requests, and infrequent update of information are some of the issues related to quality of service that consumers have to consider before using a web service.

2.7.3 Service-oriented Architecture

Service-Oriented Architecture (SOA) is an architectural style based on web services whose goal is to achieve loose coupling among interacting software components. A service is a functional process composed by a service provider to achieve desired end results for a service consumer [He03, Howard04]. Figure 2-3 shows a conceptual view of the service-oriented architecture. It consists of: client application, services interface, business objects, and data storage. A SOA "adds a services interface on top of the business objects or the legacy system that is aligned to the business processes of the organization, rather than to entities within the applications" [Bisson04]. The orchestration

layer is responsible for orchestrating calls to the business objects and managing responses with the calling client application [Bisson04, Irek03].

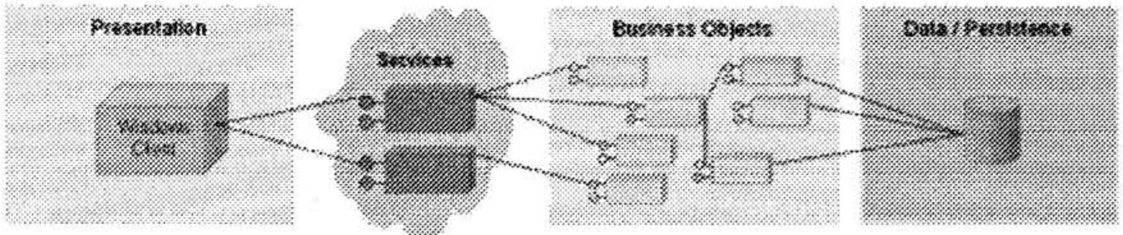


Figure 2-3 Service-Oriented Architecture [Irek03]

2.8 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a new technology for enabling the modularization of crosscutting concerns into single units called aspects, which can then be integrated with the rest of the system at join points [Bodkin02, Lee02]. An aspect file contains modular units of crosscutting implementation. Crosscutting concerns refers to the encapsulation of behaviors that affect multiple classes into reusable modules. Join points refer to locations where application classes are affected by one or more crosscutting concerns [Bodkin02, Lee02, Lesiecki02].

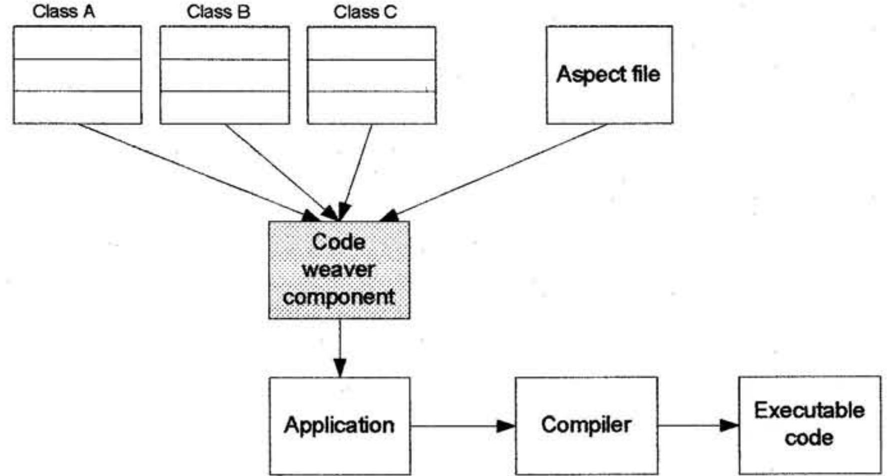


Figure 2-4 Aspect-Oriented Programming Architecture [Anastasopoulos01]

Figure 4-2 shows a conceptual overview of the weaving process between application classes and an aspect file to generate an integrated application that includes all pre-built modules of crosscutting concerns. Application classes and the aspect file are combined automatically using an integration engine, namely Code Weaver, and compiled to produce an executable source code. AspectJ [Lee02, Bodkin02] is one of the most popular tools developed specifically for AOP. It is based on JAVA language. It serves as the main engine for integrating crosscutting concerns using an aspect file.

The purpose of using AOP technology is to reduce or eliminate source code redundancy. For example, logging a method to a file for debugging, running a security check, or opening a database connection. Many classes of the application may need to use some of these methods repeatedly in different locations. Separation of concerns and source code

weaving help developers to implement these methods separately from the application, and then use the code weaver to insert them automatically at specified join points.

There are some research efforts that apply AOP in the development of software product lines [Leasint04, Loughran04a, Anastasopoulos04]. The idea of separation of concerns is used to separate optional and alternative source code from kernel source code using an aspect file. The SPL application is customized by tailoring the aspect file to include only needed optional and alternative source code. The aspect file is used along with the source code weaver to integrate variable source code with kernel source code to generate a target application.

2.9 Frame Technology

Frame technology (FT) is based on forming hierarchical reuse assemblies of framed source code [Basset97, Jarzabek03, Anastasopoulos01, Holmes03]. Source files are broken down into several hierarchical files, namely frames. The frame language composes these frames using parameterized variables and "adapt" commands. "Frames are source files equipped with preprocessor-like directives which allow parents (overlying frames) to copy and adapt children (underlying frames)" [Anastasopoulos01]. At the top level of the frame hierarchy lays a specification frame, which is used to specify children frames to be copied into parent frames at pre-defined locations.

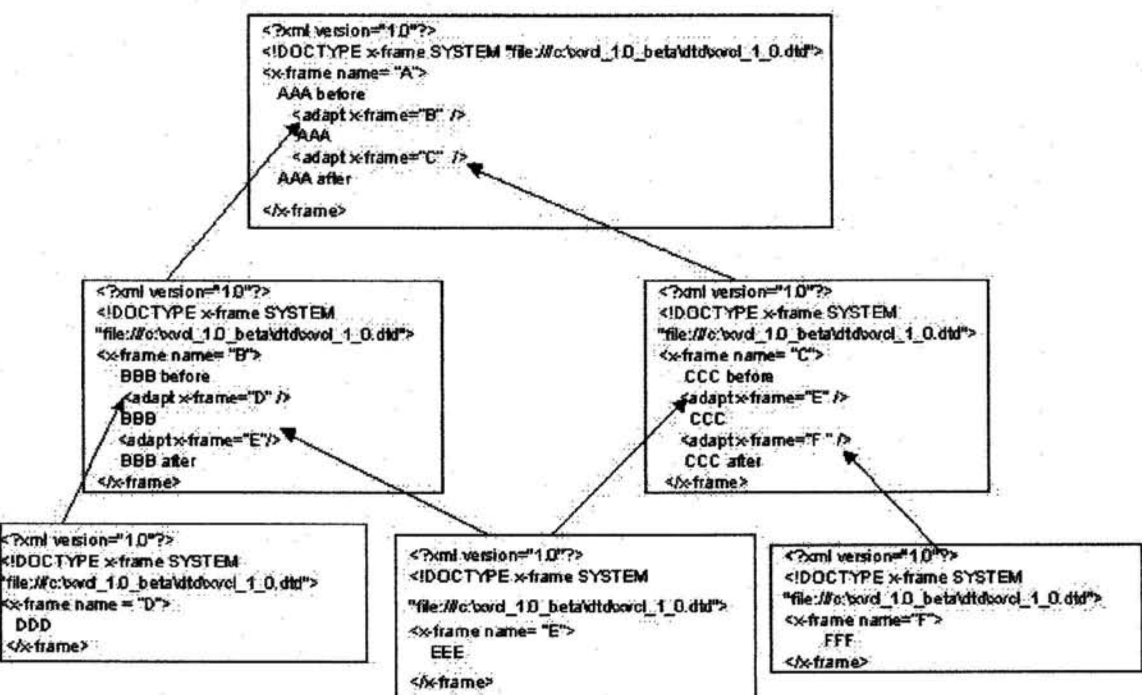


Figure 2-5 Example of an x-frame hierarchy [Zhang03b]

Figure 2-5 shows an example of an x-frame hierarchy taken from the XVCL web site [Zhang03b]. The example shows different levels of frames that are copied from children frames to parent frames using the adapt command at pre-defined locations to generate one application class file.

Some research efforts have applied Frame technology to the development of software product lines [Loughran04b, Greenwood04]. The idea of frames and SPL is used to separate optional and alternative source code from kernel source code using frames. The SPL application is customized by tailoring the top level frame (spec) to specify needed frames to be used in the composition process to generate a target application.

2.10 Summary

This chapter described related work for this research effort, which consists of different methods and environments for Software Product Line engineering. It addressed the PLUS environment in more detail than the other related work because of its close relevance to this research effort. The other methods described are: FODA, RSEB, FAST, and KobrA. This chapter also described some related technologies that are used to derive target applications from a software product line, such as aspect-oriented programming and frame technology. This research effort builds on these technologies to formulate the ideas behind the automatic customization of product lines.

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

3. PROBLEM STATEMENT AND RESEARCH APPROACH

3.1 Introduction

The purpose of this research is to investigate an approach for designing, developing, and customizing a Software Product Line for Service-Oriented Architectures (SPL-SOA). This approach builds on previous research efforts on service-oriented architectures for single systems, web services development, component-based applications, and software product lines. It also builds on two development approaches: framing technology and aspect-oriented programming, described in the literature review of Chapter 2.

The design approach is based on a multiple-view model for Software Product Lines (SPL). It addresses the engineering of an overall web service-oriented customizable software product line system where all processing activities are separated from the client application and grouped into accessible web services over the Internet.

This research also describes three different product line customization approaches for SPL-SOA architecture and implementation. The three customization approaches follow the same design architecture, but differ in the product line development and customization process, with specific consideration given to each of the customization methods described in this research.

Software product line customization is one of the major obstacles that faces SPL application engineers starting from a product line architecture and implementation. In this research, different customization methods will be described and supported by a product line independent customization prototype to help developers and application engineers configure a SPL application and generate target systems automatically from the reusable service-oriented product line architecture and components.

3.2 Problem Statement

Current approaches do not address the design, implementation, and automatic customization of software product lines based on web services. It is necessary to extend the current approaches for modeling single web services-based systems to address the unique issues of software product lines. It is also necessary to introduce an automated software development environment, based on separation of common and variable product line concerns, to allow developers to design, implement, and automatically customize web services-based software product lines to derive executable target applications.

3.3 Research Approach

This research will be based on the Evolutionary Domain Life Cycle (EDLC) and Product Line UML-based Software Engineering environment (PLUS) [Gomaa00,Gomaa04], described in Chapter 2, for developing the new concepts.

This research covers the two major phases of the EDLC:

- Software Product line Engineering phase, which includes:

- The development of a multiple-view model for web services-based software product lines.
- The development of a SPL service-oriented architecture.
- The development of reusable components

- Application Engineering. This phase covers the customization of the software product line to generate executable target systems.

The PLUS method uses the UML notation to model product line software. This research uses the PLUS method in the development of the multiple-view model for web services-based software product lines.

The following list summarizes the research activities that will be performed:

a) Develop a design approach for software product line service-oriented architecture.
b) Design three software development environments to support the automatic customization of SPL architecture and components:
    - Development of Dynamic Client Application Customization (DCAC).
    - Development of Dynamic Client Application Customization with separation of concerns (DCAC-SC).
    - Development of Static Client Application Customization with separation of concerns (SCAC).
c) Develop a proof-of-concept prototype to support the above three development environments.

d) Apply the web services-based software product line to two case studies to validate this research.

The following sections describe these research activities in more detail.

3.4 Design method for software product line service-oriented architecture

The design approach is based on a multiple-view model for Software Product Lines. The multiple-view model defines the different characteristics of a software family [Parnas79], including the commonality and variability among the members of the family [Clements02, Weiss99]. A multiple-view model is represented using the Product Line UML-based Software Engineering environment (PLUS) [Gomaa00, Gomaa04], which is extended in this research to include the design of product lines based on web services.

The design approach is illustrated by means of a case study of a hotel software product line. In this case study, a hotel product line is created for a hotel chain, which can be automatically customized to serve the needs of individual hotels.

This activity will cover the following multiple-view models:

- Use case modeling
- Feature modeling
- User interface navigation modeling
- Interaction modeling

- Activity modeling
- Software architecture modeling
- Entity class modeling
- Component interface modeling

From the above multiple-view models, certain models are addressed differently in this research to cover the unique issues related to the design of Software Product Line Web Service-Oriented Architecture (SPL-SOA): user interface navigation modeling, interaction modeling, activity modeling, software architecture modeling, and components interfaces modeling. The other models are used to complete the case study. The use case model is used to describe the functional requirements of SPL. The feature model is used to depict the kernel, optional, and alternative features in the SPL. The entity class model is used to depict the needed input when developing web services.

3.5 Development environments

This section describes software development environments to support the design and customization of Software Product Line Web Service-Oriented Architecture, in which service functionality is separated from the client side of the application and grouped into accessible web services over the Internet. The three development approaches are based on a client/server design pattern. Client applications contain only user interfaces and customizable workflows that are responsible for orchestrating web services invocation and user interface objects calls. The client application is treated as white box reuse. The

architecture and implementation are customized according to one of the three customization approaches. The server application contains all web services and database support. Web services are treated as black box reuse of services. They are either used or not used based on the customization of the client application. The three development approaches will follow the same design approach described in section 3.4. However, they will differ in how separation of concerns is used for software development and customization. The three approaches are:

1. Development of dynamic customization of client application (DCAC): Dynamic customization is defined in this research as customization of application objects at system run time. Objects are customized using a customization file that contains the target system selected features and values of parameterized variables.

2. Development of dynamic customization of client application with separation of concerns: The second development approach is an extension to the first method (DCAC) to include the separation of concerns. It is based on the dynamic customization of client applications, where objects are customized at system run time using a customization file. However, this method includes the separation of concerns, where optional and alternative source code is separated from kernel source code and placed in a variable source code file. During source code integration (referred to as code weaving), the variable source code file is used to integrate kernel source code with optional and alternative source code. The result of the integration process is a combined set of source code for the entire software

product line, including *all* optional and alternative source code. The source code for the integrated SPL system is identical to that produced by the first method (DCAC).

Separation of concerns is used to reduce complexity of developing software product lines and improve system maintenance by separating variable source code from kernel source code. Variable source code can be manipulated separately within the SPL environment and then automatically integrated with kernel source code.

3. Development of static customization of client application with separation of concerns: Static customization is defined in this research as customization of application classes at system derivation time. Classes are customized by integrating kernel source code with *only* the selected optional and alternative source code from the variable source code file. In this approach, there is no customization at system run time. The integration process is based on feature selection and an integration method that is included in the proof-of-concept prototype provided with this research. This approach is suitable for memory constrained SPL applications that require distribution of only needed target application source code.

3.6 Proof-of-concept development environment

A proof-of-concept Software Product Line Environment Prototype (SPLET) is developed to support this research. It includes the following components:

- SPL feature editor:
    - Allows SPL engineers to create a feature dependency tree and define feature relations.
    - Allows SPL engineers to create parameterized variables for each parameterized feature.
    - Allows SPL engineers to define mappings between features and related web service components.
    - Allows SPL engineers to define mappings between features and related artifacts, such as specifications, designs, and test procedures.
- Web service editor:
    - Allows SPL engineers to enter web service components and link them to their location on the Internet. The entered web service list is used by the SPL engineers to map web services to features using the feature editor component.
- Feature selector:
    - Allows application engineers to select desired features
    - Allows application engineers to enter values for parameterized variables

- Consistency checker: This component is part of the feature editor. It serves as a checker for selecting features. When a feature is selected, the consistency checker is invoked to verify selection.
- Customization file generator: This component is responsible for automatically generating a customization file that is required for the dynamic customization of client applications at system run time. It is based on the feature selector component. It sets feature selection status to true/false and stores values of parameterized variables.
- Variable source code editor: Creates a variable source code file that stores related optional and alternative source code for each feature to be used in the integration process.
- Code tracker: This component is used to locate optional and alternative source code in the variable source code file and kernel source code.
- Code weaver: This component is used for the integration process. It is responsible for integrating kernel source code with optional and alternative source code using the automatically generated variable source code file and feature selection.
- File extractor: This component is used to retrieve specifications, designs, source code, and test procedures for the selected features.

Figure 3.1 summarizes the proof-of-concept prototype SPLET.

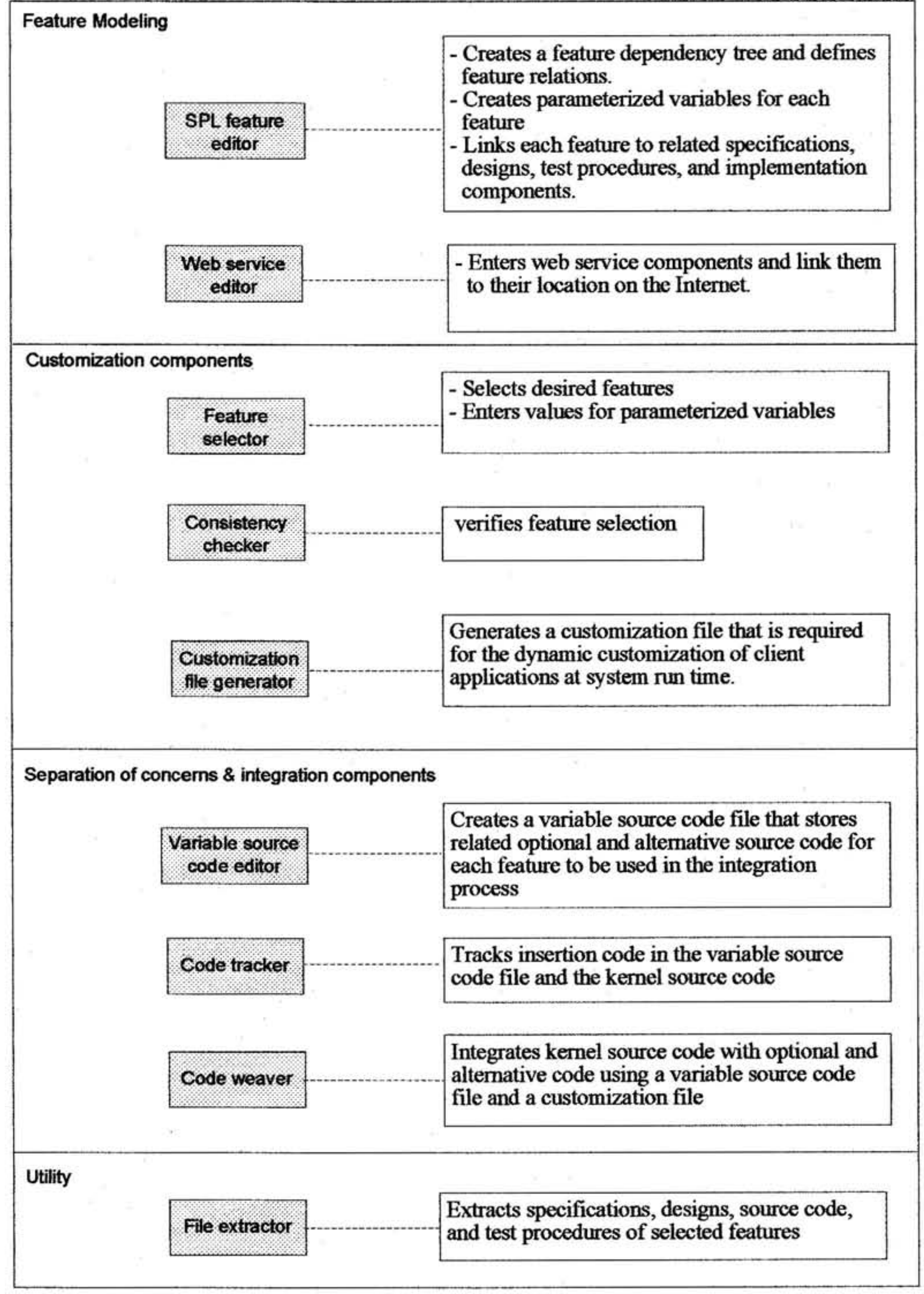


Figure 3-1 SPLET components

3.7 Validation

Apply the software product line service-oriented approach to two case studies to validate this research. The two case studies will be designed and implemented according to the proposed architecture. The two implementations will be customized to generate executable target systems. The proof-of-concept prototype SPLET will be used in the development of the environment of the two case studies and for the customization process. The two case studies are:

- Hotel system
- Radio Frequency Management System

This research's experimentations will be based on Microsoft .NET environment. Developed web services will be installed on a server with .NET framework support and Microsoft Internet Information Service (IIS). The generated target systems will be installed either on the same server or a client workstation that is connected to the server through an Extranet connection.

3.8 Comparison with other approaches

This research builds on previous research efforts. The following sections compare this research with other known research. Broadly, these research efforts can be classified into two categories: software architecture and software development approaches. The breakdown of comparisons under each category:

- Software architecture and product line research
  - Service-Oriented Architecture (SOA)
  - Component-Based Architecture (CBA)
  - web-based software product lines
  - Feature Oriented Domain Analysis (FODA)
  - Family-Oriented Abstraction, Specification, and Translation (FAST)
  - Reuse-driven Software Engineering Method (RSEB)
- Software development approaches and tools
  - Aspect-Oriented Programming (AOP) using AspectJ
  - Frames technology using XVCL
  - Knowledge Base Requirement and Elicitation Tool (KBRET)

3.8.1 Comparison with other software architectures and product line research

Service-Oriented Architecture (SOA) is an architectural style whose goal is to achieve loose coupling among interacting software components, which requires developers to design applications as collections of services [Irek03, Key04]. SOA is described in terms

of composing a single system using web services. This research builds on the concept of SOA with special concentration on the product line unique features, for the purpose of composing executable target systems.

Component-Based Architecture (CBA) is concerned with the assembly of software systems from prebuilt software components where components and frameworks have to conform to certain specifications and middleware [Bachman00]. The PhD dissertation of Ghulam Farrukh [Farrukh98] builds on the concept of CBA using configurable components in the development of software product lines. Farrukh's research presents a method, which maps a SPL model to a SPL architecture, which is then developed as a collection of reusable components and stored in a reusable library [Farrukh98]. Components of the entire SPL application are treated as black boxes. Hence the internal source code of the components is reused without any modification. This research, on the other hand, is based on software product line for SOA. It addresses the engineering of an overall web service-oriented customizable software product line system where all processing activities are separated from the client application and grouped into accessible web services over the Internet. The internal source code of client applications is treated as a white box reuse of source code. The client source code is automatically customized according to one of the customization approaches, which are included in this research. Black box reuse is used for web services components

The PhD dissertation of Mark Gianturco [Gianturco04] described a new method for modeling and generating target applications for web-based software product lines. In his

research, several web-based patterns were developed to support variability in web pages. This research also supports variability in software product lines, but differs in the customization process of target applications. Customization in Gianturco's research is based on the modification of the entity objects that are read by the visible web page objects to display variable input, text, and links to other web pages, keeping a consistent look and feel of all web pages of target applications. The customization described in this dissertation is based on feature decisions that are set using a customization file at run time for the dynamic customization approach, or the integration of kernel source code with variable source code during code binding time for the static customization approach using the concept of separation of concerns.

Feature-Oriented Domain Analysis (FODA) is a domain analysis method that is used to define a family of systems [Kang90, Cohen98]. The FODA method focuses more on structured analysis than object oriented multiple-view modeling. It includes feature models, ER diagrams, and functional models. This research builds on the PLUS method, where features are used to define a family of systems using objects-oriented analysis, design, and programming. It includes feature models, user interface navigation models, interaction models, activity models, entity class models, and component interfaces models. The multiple-view model in this research focuses on designing a SPL service-oriented auto-customizable system. This research goes further from design to implementation to cover the relationship between implementation source code and features for the purpose of customizing and deriving target systems.

Reuse-driven Software Engineering Method (RSEB) is a use case object-oriented method that is used to develop a family of related systems [Jacobson92, Jacobson97]. Variability is modeled in the use cases using variation points that include "extend" and "include" relations. Variability in use cases is introduced at these variation points. This research goes beyond use case modeling into more detailed design to support the development of customizable SPL systems. Rather than using only variation points to include or extend use cases, customization in this research is based on feature selection, where a feature can have one or more use cases.

Family-Oriented Abstraction, Specification, and Translation (FAST) approach is based on the idea of incorporating abstraction and parameterization techniques into a configuration language for modeling each member of the software product line. The configuration of each family member is mapped to templates through a source code generator [Weiss99]. Target application derivation is based on selecting needed templates and creating instances of selected templates. Templates are then manually updated to satisfy the requirement of a target application. Updated templates are then integrated with kernel source code using the source code generator. The content of templates in FAST has limited capability for customization. Templates are either selected or not selected, and selected templates have to be manually updated. Customization in this research is based on automatic adaptation to selected features at system run time using a customization file, or customization by integrating selected optional and alternative variable source code with kernel source code. This research provides a design

method and three flexible development approaches and automatic customization methods with supporting tools to enable developers and application engineers to produce highly customizable systems that do not require manual update of source code for each target system.

3.8.2 Comparison with development approaches and tools

This section compares this research with known development approaches for software product lines and their related tools that are used for customizing target systems.

Aspect-oriented programming (AOP) is a new technology for enabling the modularization of crosscutting concerns into single units called aspects, which can then be integrated with the rest of the system at join points [Bodkin02, Lee02]. AOP technology is used by several researches to define and manipulate variability in software product lines using aspect files. Aspect files contain specific source code for crosscutting concerns of variable source code. AOP has no systematic approach for creating these aspect files and selecting desired variable source code. Aspect files are created in an ad-hoc way for each target system. Keeping track of all aspect files and variable source code is very troublesome and error-prone in AOP. Also, feature related mapping of source code and features are not performed, neither consistency checks are performed based on selected features. This research provides a systematic approach in modularizing crosscutting concerns. Optional and alternative source code is grouped based on its related features in a variable source code file. Application engineers can simply select desired features and run consistency checks using the customization prototype provided

with this research, and the proper source code will be automatically integrated to generate an executable target system. There is no need for manual modification to source code to derive a target application, as it is required in AOP for product lines.

AspectJ [Lee02, Bodkin02] is one of the most popular tools developed specifically for AOP. It is based on JAVA language. It serves as the main engine for integrating crosscutting concerns using an ad-hoc aspect file. Unlike AspectJ, the SPL environment prototype (SPLET) creates a complete environment for the SPL application. All variable source code is contained in SPLET by associating variable source code with the SPL system features. Optional and alternative source code is automatically integrated with kernel source code using SPLET's code weaver component by selecting target system features and applying consistency checks. Also, SPLET is designed to support most popular languages such as C++, C#, JAVA, J++, and visual Basic.

Frame technology (FT) is based on forming hierarchical reuse assemblies of framed source code [Basset97, Jarzabek03, Anastasopoulos01, Holmes03]. Source files are broken down into several hierarchical files, namely frames. The frame language composes these frames using parameterized variables and "adapt" commands. In an object-oriented application, frame files can grow large in number and become very difficult to manage and maintain. Similar to AOP, frames do not describe how to map features to source code, and do not provide consistency checks to verify whether a set of frames is consistent with the SPL model. This research provides a systematic approach

for relating source code to features for the purpose of automating customization of SPL applications.

XVCL [Hongyu03] is one of the recently developed tools for frame technology in the SPL domain. It serves as an engine for integrating frames together based on pre-defined variables. Similar to AspectJ, it does not cover the SPL life cycle and has no automation to select integrated code.

Knowledge Base Requirement and Elicitation Tool (KBRET) [Gomaa92, Gomaa96a] was developed to support feature selection of SPL systems and apply consistency checks to verify selections. Feature navigation in KBRET does not show the overall breakdown of the system features and related designs and implementation components. SPLET enables designers, developers, and application engineers to visualize the entire SPL system more conveniently. SPLET provides a facility to see and extract all analysis, designs, code, and test procedures for each feature separately. It also provides a facility to execute web services components automatically for functional testing of components without leaving the environment. SPLET includes a facility to create separation of concerns between optional and alternative source code from kernel source code and an integration engine for automatic generation of executable target systems by selecting target system features and applying consistency checks.

3.9 Summary

This chapter has addressed the problem statement for this research and explained the research approach and the breakdown of tasks to be performed. It also provided an overall comparison with related software design architectures and different development approaches including their related tools. The design approach will be described in the next chapter.

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

4. A DESIGN METHOD FOR SOFTWARE PRODUCT LINES BASED ON WEB SERVICES

4.1 Introduction

This chapter describes the software product line modeling approach for product lines based on Web Service-Oriented Architectures. The Evolutionary Software Product Line Engineering Process (PLUS) [Gomaa96, Gomaa99, Gomaa04] is used to show the major activities performed in the development of the proposed Software Product Line (SPL) based on Web Services.

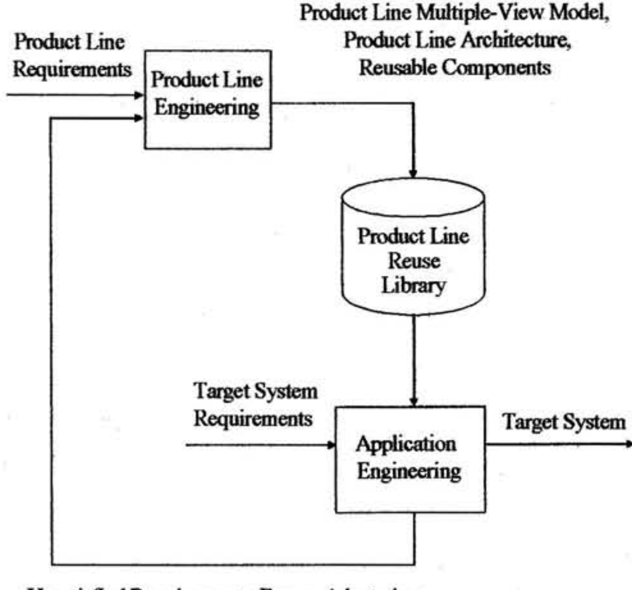


Figure 4-1 Evolutionary Software Product Line Engineering Process

The Evolutionary Software Product Line Engineering Process consists of two main phases, as shown in Figure 4-1:

c) Software Product line Engineering. A product line multiple-view model, which addresses the multiple views of a software product line, is developed. The product line multiple-view model, product line architecture, and reusable components are developed and stored in the product line reuse library.

d) Application Engineering. A target system is a member of the software product line. The multiple-view model for a target system is configured from the product line multiple-view model. The user selects the desired features for the product line member (referred to as target system). Given the target system features, the product line model and architecture are adapted and tailored to derive the target system model and architecture. The architecture determines which of the reusable components are needed for configuring the executable target system.

Earlier papers and researches have described how this approach was carried out before [Gomaa96, Gomaa99] and after the introduction of the UML [Gomaa02, Gomaa04]. This chapter describes how product line engineering can be carried out for product lines based on Web Services.

This chapter covers the design of the SPL Engineering Phase based on Web Service-Oriented Architectures. The design, implementation, and configuration approach mentioned in this research is focused on SPL for a chain of systems that belong to a

single organization, for example, a chain of Hilton hotels or Avis car rentals, where many branches are distributed in different locations. Each branch will have a software system customized to its needs based on available facilities. This research addresses the engineering of an over all Web Service-Oriented customizable software product line system where all functional activities are separated from the client application and grouped into accessible Web Services over the Internet. The customizable client application contains:

- Customizable navigation screens
- Events workflows.

Both contents are described in detail later in this chapter.

Once the Software Product Line is designed and developed, target systems are customized through the domain independent customization prototype, described in Chapter 6, for deciding which screen to display, which web service to invoke, and what parameterized variables to use, based on selected features.

4.2 Design Architecture of SPL Engineering Phase

The design architecture is based on a multiple-view model for Software Product Lines. The multiple-view model defines the different characteristics of a software family [Parnas79], including the commonality and variability among the members of the family [Clements02, Weiss99]. A multiple-view model is represented using the UML notation [Rumbaugh99, Gomaa00] and considers the product line from different perspectives.

This section describes the software product line modeling approach for product lines based on Web Services. In particular, the multiple-view modeling approach described in Chapter 3 needs to be tailored for modeling product lines based on Web services. The method is described by means of a hotel software product line (SPL), which is used as an example of applying the software design method for software product lines based on web services. In this example, a Hotel product line is to be created for a hotel chain, which can be customized to the needs of individual hotels.

4.2.1 Use Case Modeling

Figure 4-2 depicts the Use Case diagram for the Hotel SPL, which captures the overall software requirements. The Use Cases in Figure 4-2 are categorized as kernel, optional, or alternative as given by the PLUS environment [Gomaa04]:

- Kernel: Use case that exists in all members of the product line.
- Optional: Use case that may or may not exist in a given product line member.
- Alternative: One of a group of alternative use cases is selected for a given product line member.

The actors for this use case model are the users of the product line, providing inputs to a product line member system and receiving outputs from it.

- **Reservation Clerk** – Performs actions pertaining to room reservation.
- **Front Desk Clerk** – Performs duties pertaining to check-in and checkout of hotel rooms and walk-in reservations.
- **Manager** –Updates hotel prices and request management reports.

- **Restaurant Staff** – Add restaurant charges to guests' billing records.
- **Timer** – Controls the initiation of periodic hotel functions at a predetermined time.

Briefly, the use cases are:

- **Make Room Reservation:** A reservation or check-in clerk makes reservations for one or more rooms. Users will also be able to cancel reservations, update reservations, query reservations, and verify customer credit cards.
- **Make Residential Reservation** As an alternative to room reservation, a hotel can consist of residential suites, where a guest can occupy a suite for a month at a time, paying a monthly rate. A guaranteed reservation is required for residential suites, with payment made on the first day of the month.
- **Make Block Reservation:** A travel company can book a block of rooms for their customers. The travel company will be billed directly instead of billing customers individually. Check in and check out will be made for the reserved block.
- **Check-in Single Customer:** The front-desk clerk checks in guests with single room reservations.
- **Check-in Block Customer:** The front-desk clerk checks in guests with block reservations.
- **Checkout Single Customer:** The front-desk clerk checks out guests with single reservations.

- **Checkout Block Customer:** The front-desk clerk checks out guests with block reservations.

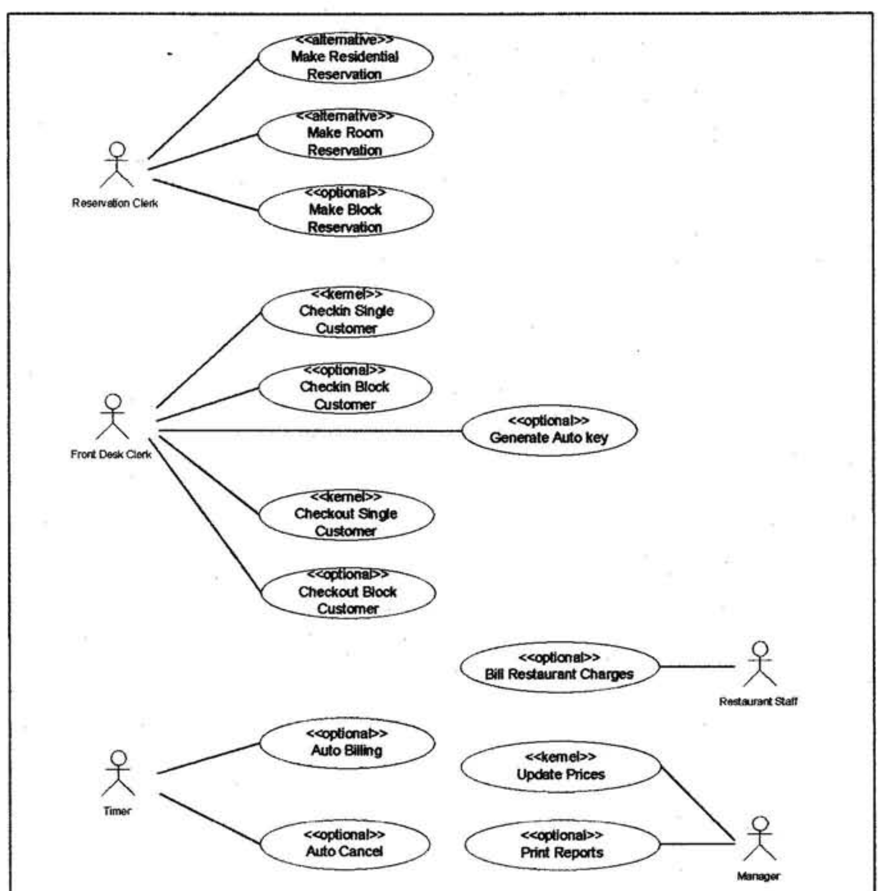


Figure 4-2 Use Case Diagram

- **Generate Auto Key:** Some hotels may use electronic cards for door keys rather than regular door keys. Assigned room numbers will be encoded on the electronic cards.
- **Auto Billing:** At a pre-specified time, bills customers who have guaranteed reservations and do not show up.

- **Auto Cancel:** At a pre-specified time, cancels non-guaranteed reservations.
- **Bill Restaurant Charges:** Restaurant charges are added to guest billing record.
- **Update prices:** Allows managers to update room prices.
- **Print Reports:** Allows managers to request reports such as: No-show reports, reservation reports, financial reports.

4.2.2 Feature Modeling

A feature dependency model is derived from the use case model. Product line features are categorized as kernel, optional, or alternative features. By selecting the features required for a given member of the product line, an application can be derived from the product line. Related features are grouped together into feature groups. The possible feature groups are:

- *Mutually exclusive groups*: Zero or one feature can be selected out of a group of features.
- *Exactly one of feature group*: One and only one feature can be selected out of a group of features.
- *Zero or more of feature group*: Zero or more features can be selected out of a set of features.
- *Mutually inclusive group*: If one feature is picked the other feature(s) in the group must be picked.

The feature model (Fig. 4-3) depicts the features, feature dependencies, and feature groups for the hotel product line. In this model, "RoomReservation" and

"ResidentialReservations" are two alternative features grouped under an *exactly-one-of-feature-group*, where the "BlockReservation", "AutoCancel", and "AutoNoShowbilling" optional features depend on the "RoomReservation" alternative feature. If "Residential Reservation" is selected instead, the above optional features would not be available for the derived application.

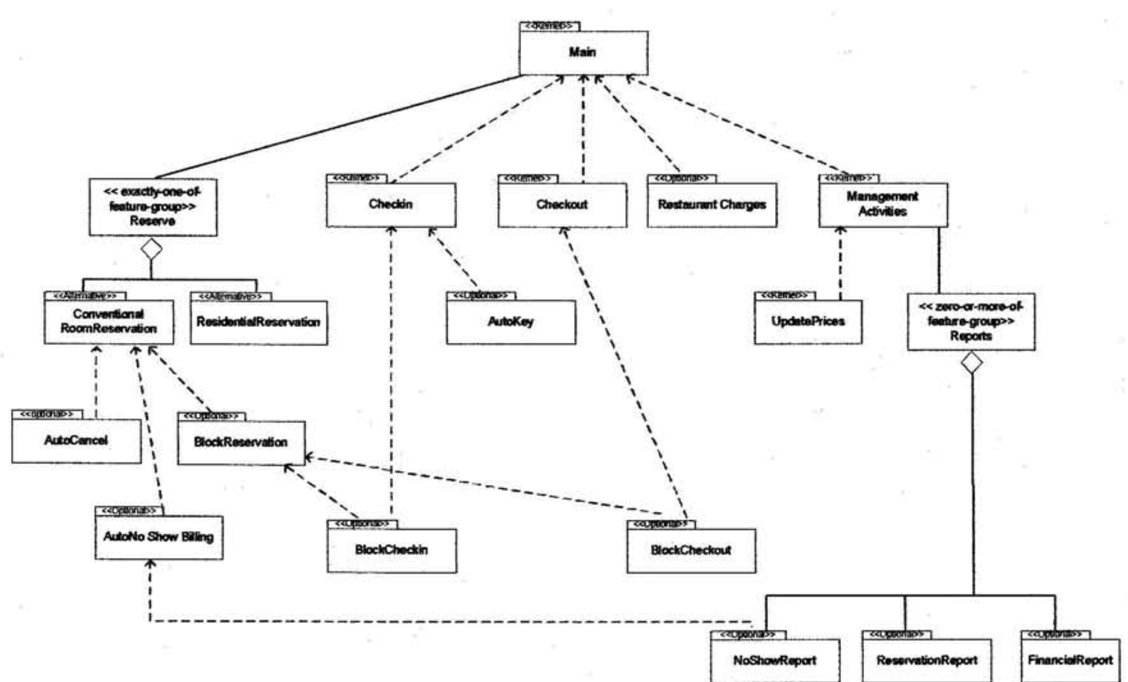


Figure 4-3 Feature Dependency Model

4.2.3 User Interface Navigation Modeling

Since this design method is based on a service-oriented architecture for the product line, it is important to show the navigation between user interface screens. Each user interface screen is supported by a user interface object, which is in turn associated with one or more Web services. Each user interface object contains a GUI and a customizable

workflow for members of the software product line. The GUI will be responsible for accepting user input and user requests to initiate events that are translated into method calls to web services. After receiving the user input, the user interface object interacts with the appropriate Web service.

Figure 4-4 shows the system navigation from the user perspective. Each user interface screen is supported by a user interface class, which is categorized as kernel, optional, or alternative. Each class is depicted with two stereotypes, the role stereotype is <<user interface>> and the reuse stereotype, such as <<kernel>> or <<optional>>. The navigation model depicts the user interface classes that can be accessed from a given user interface class. For example, from the Main Reservation user interface, the Room Reservation user interface can be reached. Based on the features desired for a given product line member, all kernel classes will be selected, some of the optional classes will be selected, and a choice is made among alternative classes. Each user interface object interacts with relevant web services, which is shown in the dynamic model.

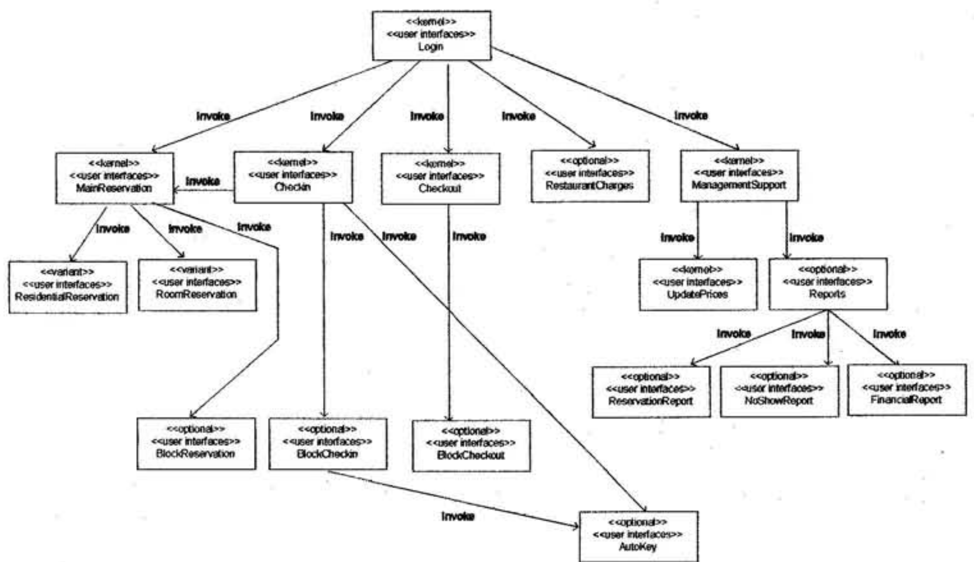


Figure 4-4 User Interface Navigation Model

Figure 4-5 shows a sample GUI for the "RoomReservation" user interface class, one of the classes that is depicted on the navigation model in Figure 4-4.

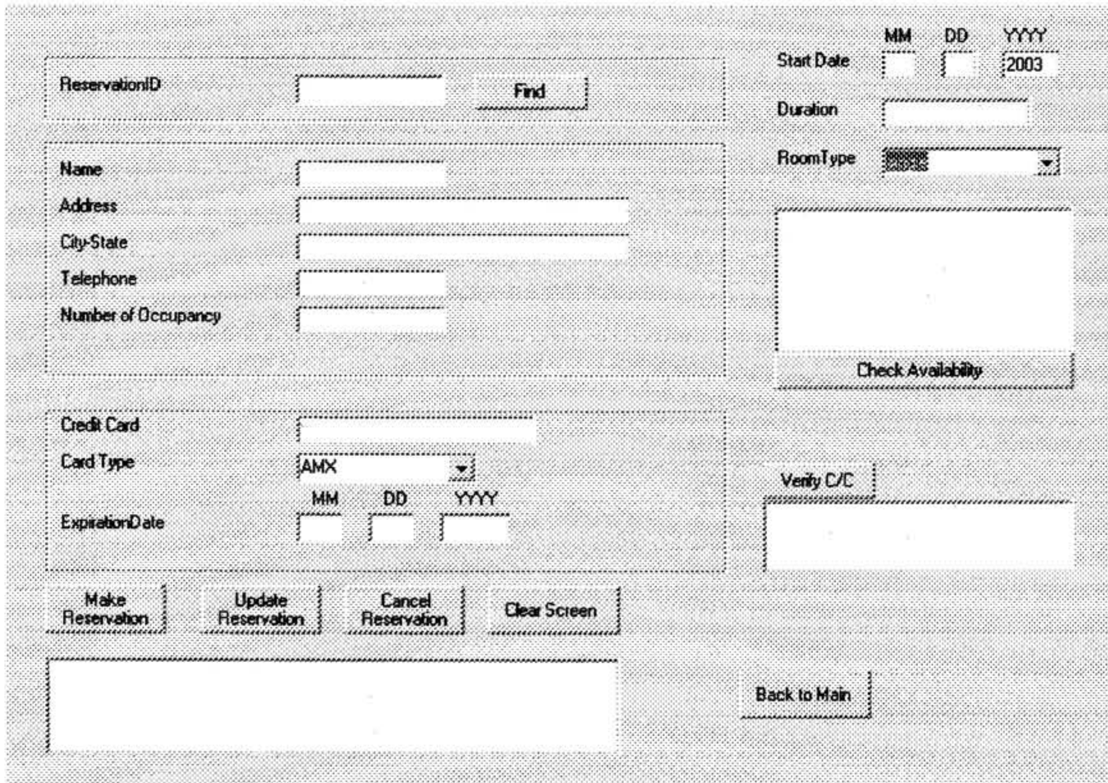


Figure 4-5 GUI –RoomReservation UI

4.2.4 Interaction Modeling

Next, the interaction between the user interface object, described in the previous section, and the appropriate Web service is modeled. This section describes the interaction between the user interfaces and web services. Figure 4-6 is a collaboration diagram for "RoomReservation" user interface object for processing a reservation for a single room. Since the core functionality is encapsulated in Web services classes, the collaboration diagram shows the interaction at a high level. To reserve a room, the system requires 3 web services: AvailabilityWS, CreditWS, and ReserveRoomWS. The user interface

object accepts the guest's information and directs the input to the appropriate web service as shown in Figure 4-6.

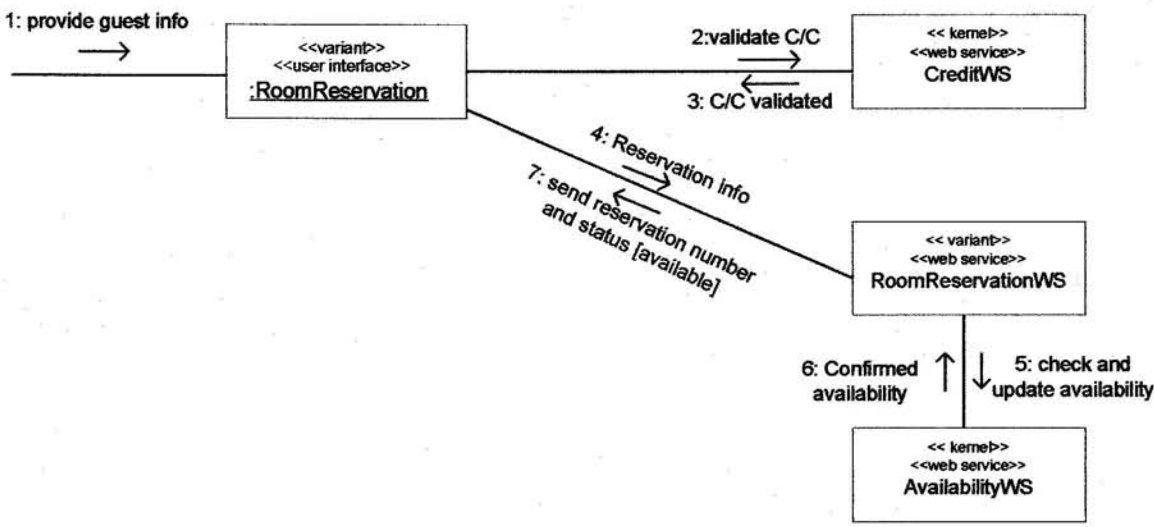


Figure 4-6 Collaboration Diagram – Reserve single room

Figure 4-7 provides more detail describing the object interaction within each web service and between web services and user interface.

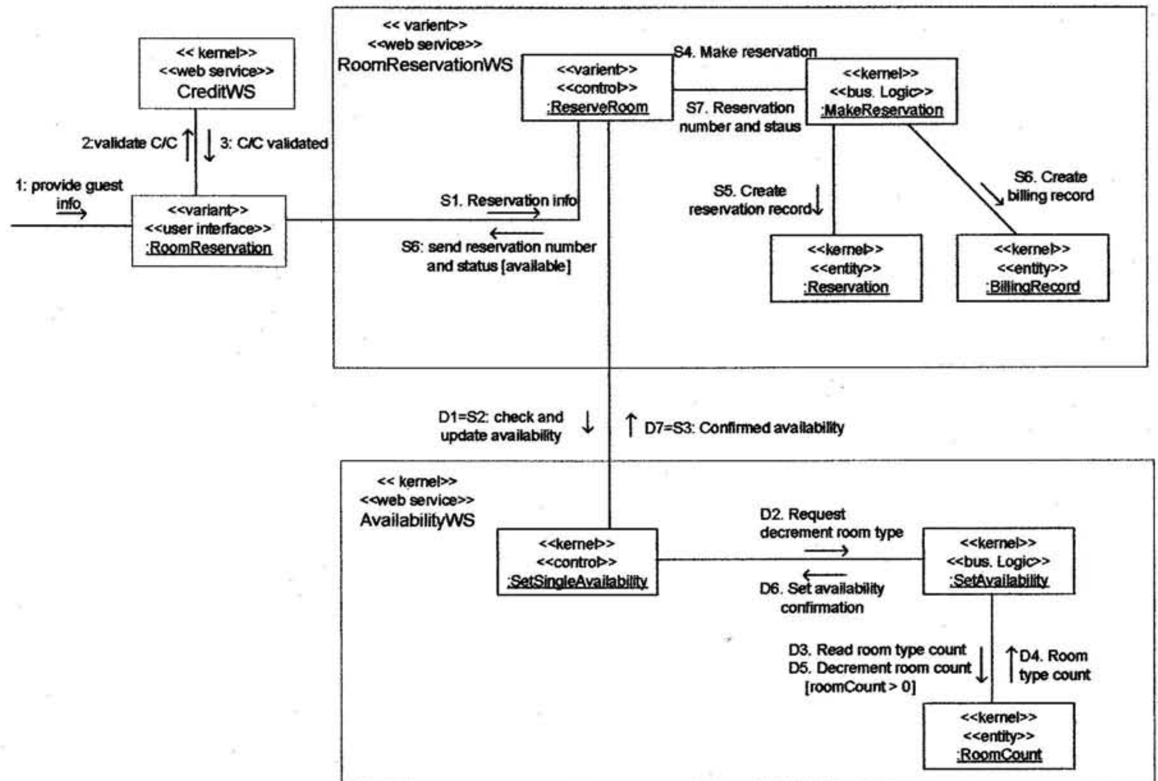


Figure 4-7 Expanded Collaboration Diagram – Reserve single room

4.2.5 Activity Modeling

The activity diagrams, in the SPL Service-Oriented approach, describe the workflow of each event initiated by the user. Each user interface object is associated with one or more workflows. Workflows have two major tasks:

- Invoke web services: Workflows show the sequence in which web services methods are called for processing a complete event.
- Invoke other user interfaces: Workflows show the navigation pattern in which other user interfaces are invoked.

The workflow for the SPL Service-Oriented architecture is customized during target system configuration. Figure 4-8 shows a customizable activity diagram for the "MainReservation" user interface. "ResidentialReservation" UI and "Room Reservation" UI are mutually exclusive alternatives where only one of them can be invoked by the user. During customization, a path will be selected for the application to identify which GUIs or web services will be invoked. Feature conditions are used for this purpose. For example, [feature = RoomReservation] and [feature = ResidentialReservation] are two feature conditions used in the activity diagram of Figure 4-8 to show the mutually exclusive feature decisions in the workflow. The customization of workflows is described in detail in Chapter 5.

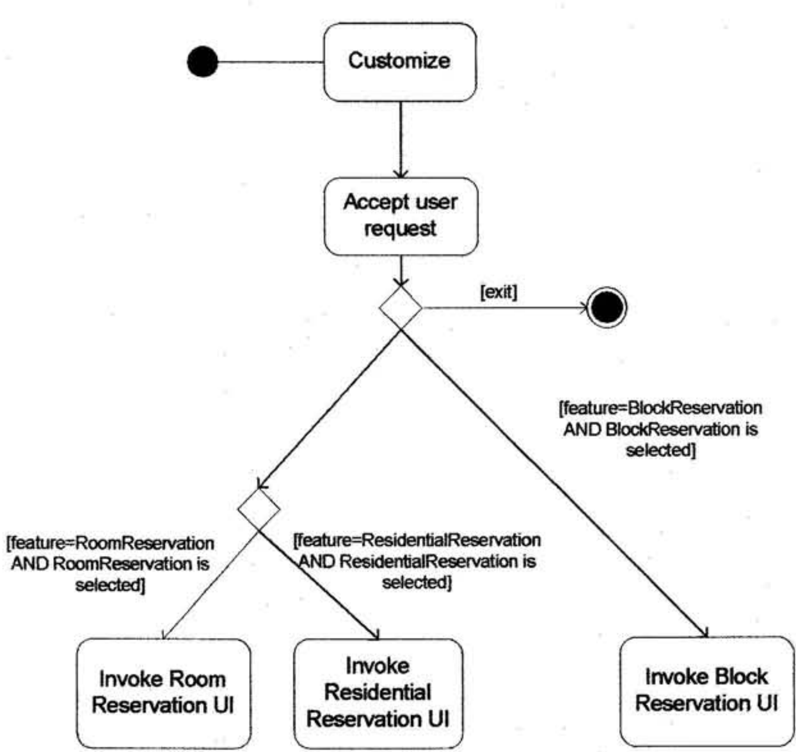


Figure 4-8 Activity Diagram– Main Reservation

Figure 4-9 is an overall activity diagram for the "RoomReservation" UI. It shows all the possible events that can be initiated by a user and all web service method calls.

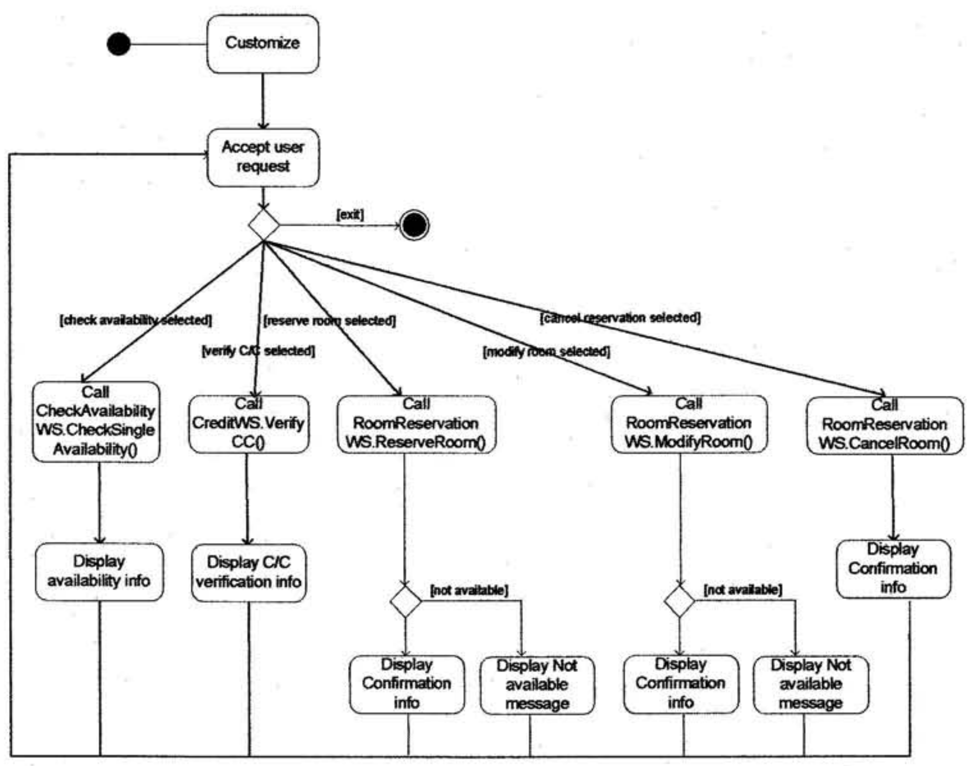


Figure 4-9 Activity Diagram – Overall Room Reservation UI

Figure 4-10 shows a sample workflow for processing a single room reservation. The activity diagram in this figure shows the workflow and required web service methods for reserving a single room. Once the front desk clerk verifies the guest's credit card by calling the "CreditWS" web service, reservation can be made by calling the "ReserveRoomWS" web services.

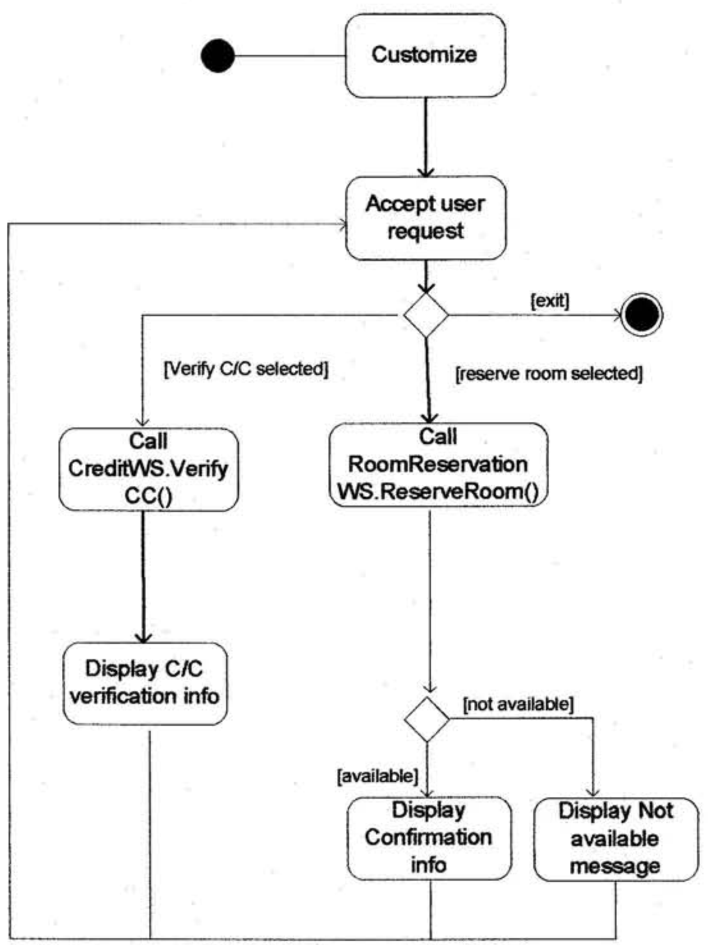


Figure 4-10 Activity Diagram–Reserve Room

4.2.6 Software Architecture Modeling

4.2.6.1 Web Services

From the activity modeling, all possible service requests are identified. These services are organized and grouped into related web services based on their objects interaction, described in section 4.2.4 - Interaction Modeling. Figure 4-11 shows a sample grouping of methods into Web Services. For example, ReserveRoom Web Service contains these related methods: Reserve Room, Cancel Room, Modify Room, Check-in Room, and

Check-out Room. These methods may share internally some of the web service objects. However, all objects are hidden within each web service. They can not be inherited by other web services. For example, the entity class RoomCount is used by ReserveRoom, ModifyRoom, and CancelRoom methods in the RoomReservationWS web service. The RoomCount entity is used to store room availability information. The above methods update this entity using different business logic objects.

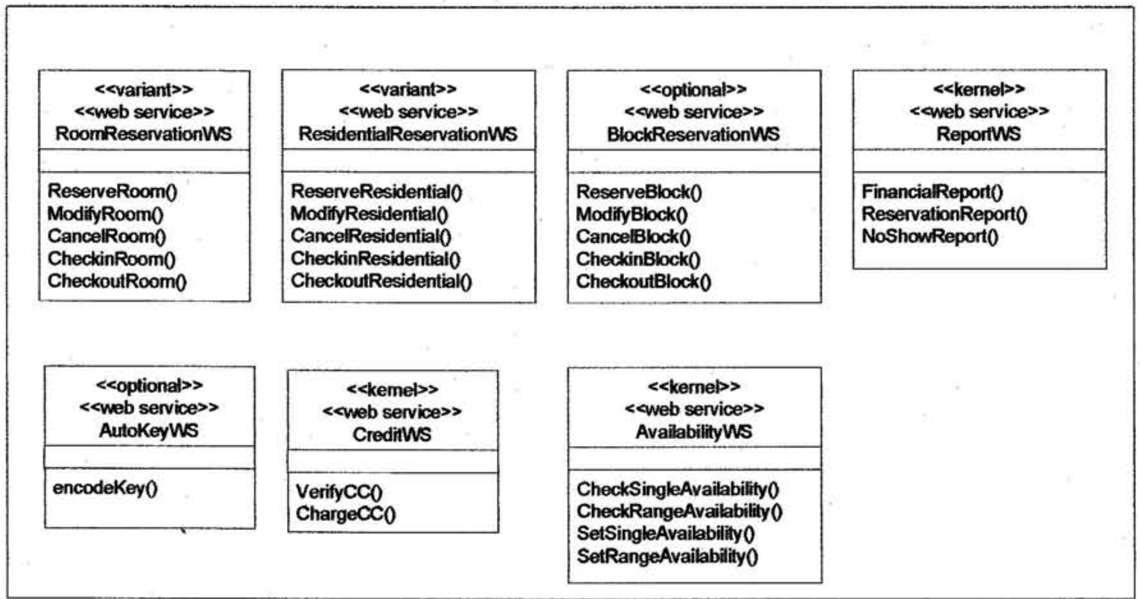


Figure 4-11 Example of Web Services grouping

Web Services Methods are developed according to the specified design. A Web Service encapsulates the implemented methods as a black box, hiding all internal activities from the outside world. These black boxes use the same XML/SOAP technology to interface with outside applications. Variability is handled by the client application according to the

customizable workflows. During customization of client application, workflows are customized to determine which web service to invoke.

4.2.6.2 Web Service Input/Output

Since all interactions between user interfaces and Web services rely on message communication, it is very important to specify all inputs to and outputs from each method of the Web service. Figure 4-12 gives sample input/output for three of the Web methods of the ReserveRoomWS Web service.

Method	Input	Type	Output	Type
ReserveRoom()	Name Address Tel CreditCardNo ExpirationDate CreditType RoomType ArrivalDate NumberOfDayes NumberOfOccupancy	string string int int date string string date int int	ReservationNumber ReservationStatus	int string
ModifyRoom ()	Name Address Tel CreditCardNo ExpirationDate CreditType RoomType ArrivalDate NumberOfDays NumberOfOccupancy	string string int int date string string date int int	Confirmation	int string
CancelRoom ()	ReservationNumber	string	Confirmation	string

Figure 4-12 Sample Input/Output for ReserveRoomWS

4.2.7 Attributes of Entity Classes

An important part of modeling Web services is to capture the attributes of the entity classes, which are information intensive. The collaboration diagram on Figure 4-7 depicts three entity objects. The entity classes and their attributes are depicted in Figure 4-13, which is the information needed for completing a reservation transaction and recording the information in the database.

<<kernel>> <<entity>> Reservation
Name: string Address: string Tel: string CreditCardNo: int ExpirationDate: date RoomType: string ArrivalDate: Date NumberOfDays: int NumberOfOccupancy: int

<<kernel>> <<entity>> BillingRecord
ReservationNumber: int ChargeType: string Charge: int

<<kernel>> <<entity>> RoomCount
RoomType: string NoOfRooms: int Date: date

Figure 4-13 Sample Entity Attributes for ReserveRoomWS

4.2.8 Design of Component Interfaces

In developing the software architecture, the objects from the interaction model are now designed as components in terms of their interfaces and interconnections using the UML 2.0 structured class notation. Components communicate with each other through ports, which support provided and/or required interfaces. Figure 4-14 shows an example of how ports and connectors facilitate component interactions. Components are categorized

(using UML stereotypes) to show the kernel, variant, or optional components for the product line. The "RoomReservation" user interface component has two required ports (consisting of required interfaces): RVerify and RReserve, which are used to connect it to the "CreditWS" and "RoomReservationWS" web service components. Figure 4-14 also shows the connection between two different web service components, "RoomReservationWS" and "AvailabilityWS".

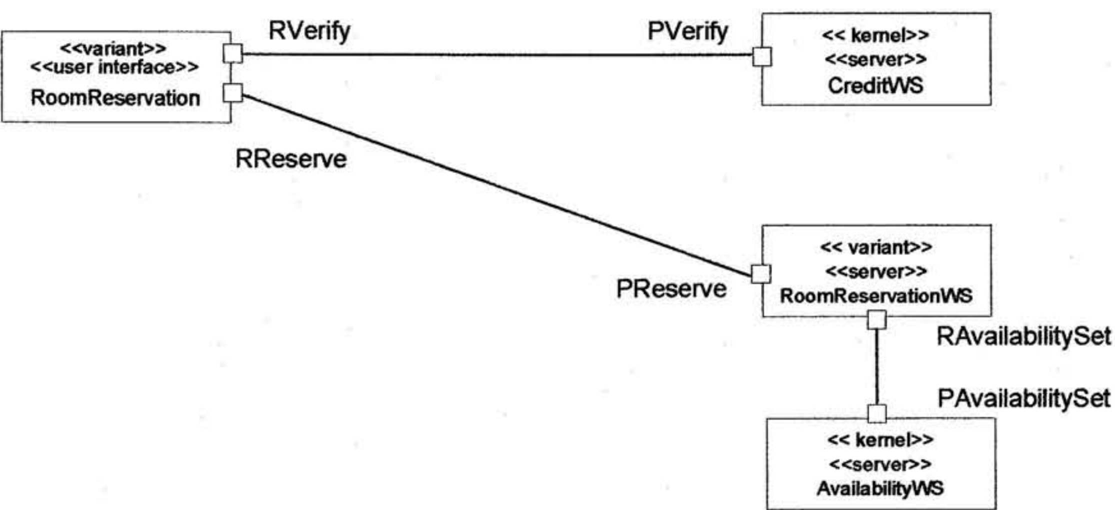


Figure 4-14 Example of ports and connectors - RoomReservation Feature

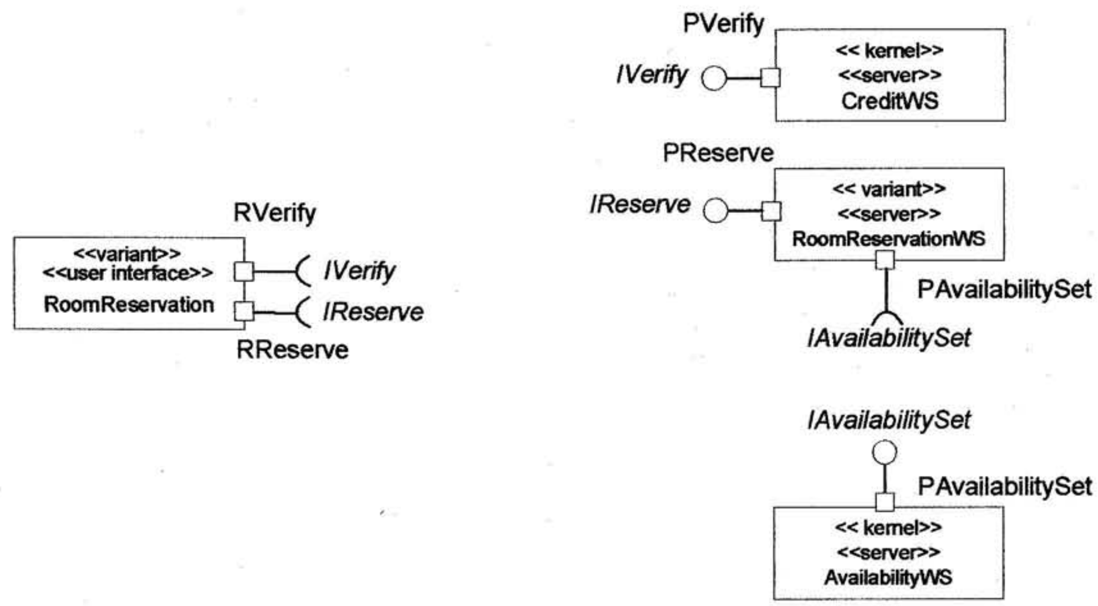


Figure 4-15 Example of ports, provided, and required interfaces

Figure 4-15 defines the *required* and *provided* interfaces for the interfaces shown in Figure 4-14. The interfaces between user interface and web service components are designed using a client/server pattern such that a user interface component always requires a port provided by a web service component, while a web service component always provides a port for the user interface component. The interaction between web service components or between user interface components can result in a component having both provided and required ports.

Figure 4-16 shows the interfaces of components using the UML static modeling notation. This design depicts the provided web service methods for each interface. Web service

methods are invoked based on the customized workflows of the product line, as described earlier in the customizable activity diagram (Section 4-2-5).

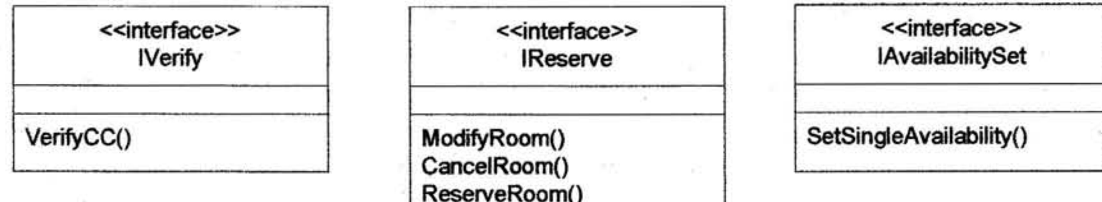


Figure 4-16 Example of port interfaces design

4.3 Summary

This chapter described the software product line modeling approach for product lines based on Web Service-Oriented Architecture where all functional activities are separated from the client application and grouped into accessible web services. The design architecture is based on a multiple-view model for Software Product Lines. The multiple-view model defined the different characteristics of a software family [Parnas79], including the commonality and variability among the members of the family [Clements02, Weiss99]. A multiple-view model was represented using the UML notation [Rumbaugh99, Gomaa00] and considered the product line from different perspectives. The method was described by means of a hotel software product line (SPL), which was used as an example of applying the software design method for software product lines based on web services. In the example, a hotel product line was created for a hotel chain, which could be customized to the needs of individual hotels.

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

5. DEVELOPMENT APPROACHES FOR PRODUCT LINE CUSTOMIZATION AND SEPARATION OF CONCERNS

5.1 Introduction

This chapter describes three different approaches to develop a Software Product Line Web Service-Oriented Architecture and implementation, where all service activities are separated from the client application and grouped into accessible web services over the Internet. The three development approaches are based on a client/server design pattern specific to Service-Oriented Architecture (SOA). Client applications contain only user interfaces and customizable workflows that are responsible for orchestrating web services invocation and user interfaces calls. Server applications contain all web services and database support. The three development approaches follow the same design method, described in chapter 4, but differ in the customization process. The three approaches are:

- Dynamic customization of client application (DCAC). Dynamic customization is defined in this research as customization of application objects at system run time. Objects are customized using a customization file that contains the target system selected features and values of parameterized variables.
- Dynamic customization of client application with separation of concerns (DCAC-SC). The second development approach is an extension to the first method by incorporating the separation of optional and alternative feature source code from

kernel source code at product line development time, and the integration of *all* separated source code with kernel source code at customization time.

- Static customization of client application with separation of concerns (SCAC). Static customization is defined in this research as customization of application objects at system customization time. Objects are customized by integrating kernel source code with *only* selected optional and alternative source code producing the exact source code needed for running a single target system.

The three development and customization approaches are based on the Software Product Line Environment for Service-Oriented Architecture (SPLE-SOA) that is provided with this research. This chapter starts by describing the first development approach (DCAC) in section 5.2. Section 5.2.1 applies the DCAC approach to the hotel system case study. Sections 5.2.2 and 5.2.3 list the advantages and disadvantages of using the DCAC approach. Section 5.3 introduces the issue of separation of concerns, which is used in the next two development approaches (DCAC-SC and SCAC). Section 5.4 describes the second development approach (DCAC-SC). Section 5.4.1 applies the DCAC-SC approach to the hotel system case study. Section 5.4.2 lists the advantages and disadvantages of using the DCAC-SC approach. Section 5.5 describes the third development approach (SCAC). Section 5.5.1 applies the SCAC approach to the hotel system case study. Sections 5.5.2 and 5.5.3 list the advantages and disadvantages of using the SCAC approach. Section 5.6 compares the three development approaches. Section 5.7 describes the usage of each approach. Section 5.8 summarizes this chapter.

5.2 Dynamic customization of client application

The first development approach is based on the dynamic customization of the client application, where objects are customized at system run time using a customization file that contains the target system selected features and values of parameterized variables. Figure 5-1 shows a conceptual overview of the approach. It consists of the customizable SPL system architecture and the SPL environment.

The customizable SPL system architecture in Figure 5-1 is based on the client/server design pattern, where the client application contains only user interface objects and a customizer object, and the server application contain all web services and database support.

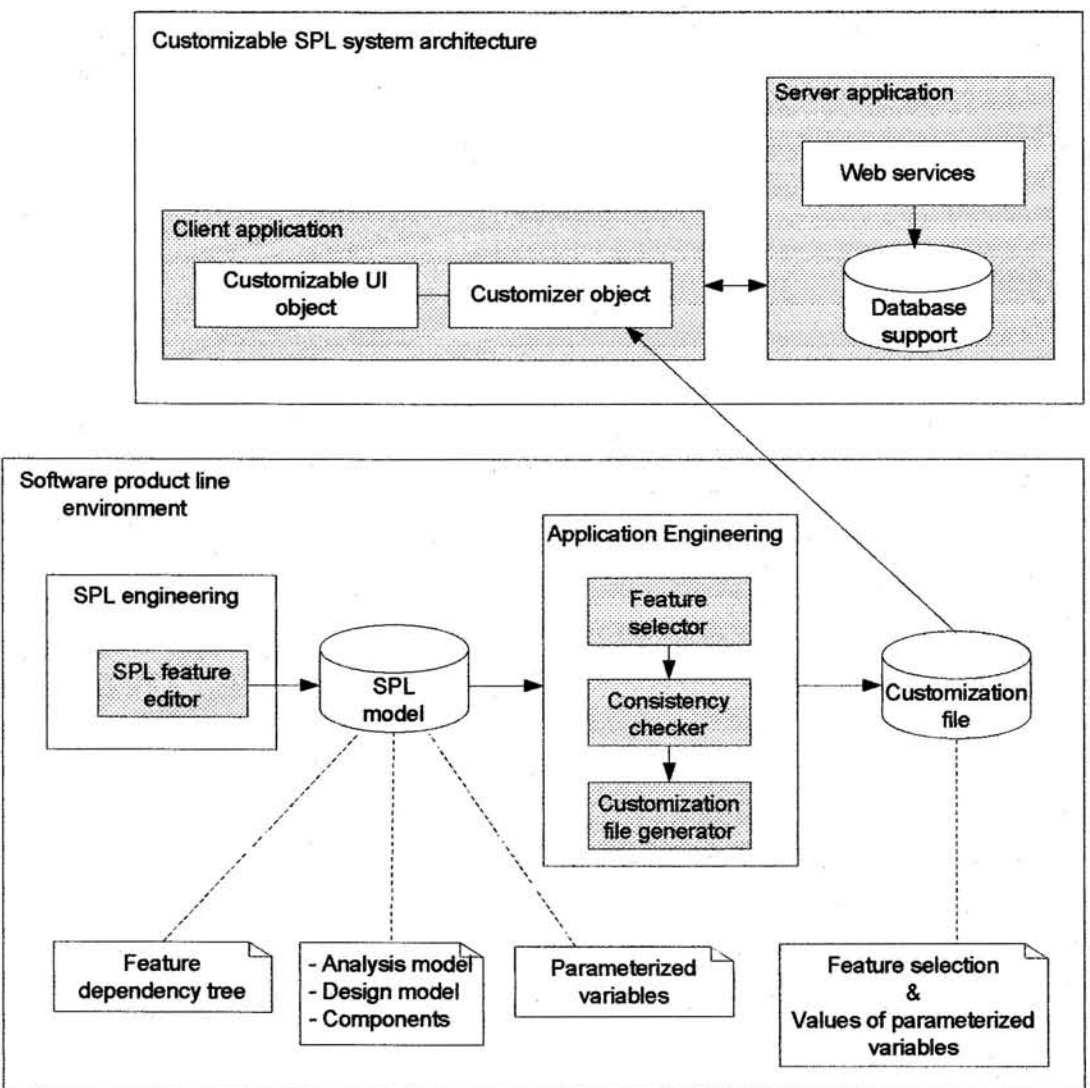


Figure 5-1 Conceptual overview of DCAC approach

The software product line environment in Figure 5-1 shows a conceptual overview of the approach from the SPL engineering phase to the application engineering phase (SPL customization). The overall life cycle is based on the PLUS method [Gomaa00,Gomaa04], which includes the following steps:

- SPL Engineering:
  - Analyze SPL customizable system
  - Design SPL customizable system
  - Implement SPL customizable system
- Application Engineering (SPL customization):
  - Use the feature selector to select desired features and apply consistency checking rules, described in Chapter 6.
  - Store target system customization information in the customization file using the customization file generator. The customizable application will read this file to customize user interface objects and their workflows, described in detail in the Dynamic Client Application Customization Pattern in Figure 5-2.
- Deploy the customizable SPL system and related web services.

The customizable SPL system uses the customization file produced in the application engineering phase to customize a target system at run time. The customizer object (Fig. 5.1) reads the customization file and stores all customization information in the customizer object's local storage (arrays, data table, etc.) to be used for customizing the client application user interfaces and their workflows. User interfaces are customized by enabling or disabling buttons, and by setting appropriate display variables. Workflows are customized by tailoring decisions on which user interface to call or which web

service to invoke. This approach is described in the Dynamic Client Application Customization (DCAC) Pattern in Figure 5-2.

The activities specific to this research relate to each phase in Figure 5-1 and are as follow:

- SPL engineering phase:
    - SPL feature editor: Allows users to create a feature dependency tree and define feature relations, create parameterized variables for each feature, and link each feature to related specifications, designs, test procedures, and implementation components.
- Application engineering phase:
    - Feature selector: Allows users to select desired features, and to enter value of parameterized variables.
    - Consistency checker: This component is part of the feature selector. It serves as a check for selecting features. When a feature is selected, the consistency checker is invoked to verify selection by consulting the feature dependency model for inconsistent feature selection.
    - Customization file generator: This component is responsible for generating a customization file that is required for the dynamic customization of client applications at system run time.

The DCAC approach is described next as an architecture pattern. It provides a detailed description of the development approach, which can be applied to any SPL application based on web services.

Dynamic Client Application Customization Pattern

Intent

Provide a consistent reusable solution to the implementation architecture of a client/server software product line using web services with provision for *dynamic* client application customization.

Motivation

The goal of developing software product lines is to promote flexible software reuse. With the introduction of web services to SPLs, there is a need for developing a systematic approach that enables developers to implement a customizable system that can be dynamically customized into many single target systems without the need to modify any of the source code. Using the feature selector component, user interfaces and workflows of SPL systems can be automatically adjusted at *run time* to serve a single target system.

Solution

The idea behind the (DCAC) pattern is the development of dynamic client application that can be customized at system run time.

The DCAC Pattern has two main steps:

1. SPL Customization
2. Target application interaction

Step 1: SPL Customization

This step involves selecting desired optional and alternative features to be included in the target system. The feature selector component provides a facility to make feature selection from a SPL model and run consistency checks to verify selections. Once features are selected, selection information will be stored in the customization file by the customization file generator. The dynamic client application is customized by reading the customization file at run time.

Components description:

- Feature selector: Allows users to selects desired features, and allows entry for parameterized variable values.
- Consistency checker: Verifies feature selection.
- Customization file generator: Generates a customization file for each target system.

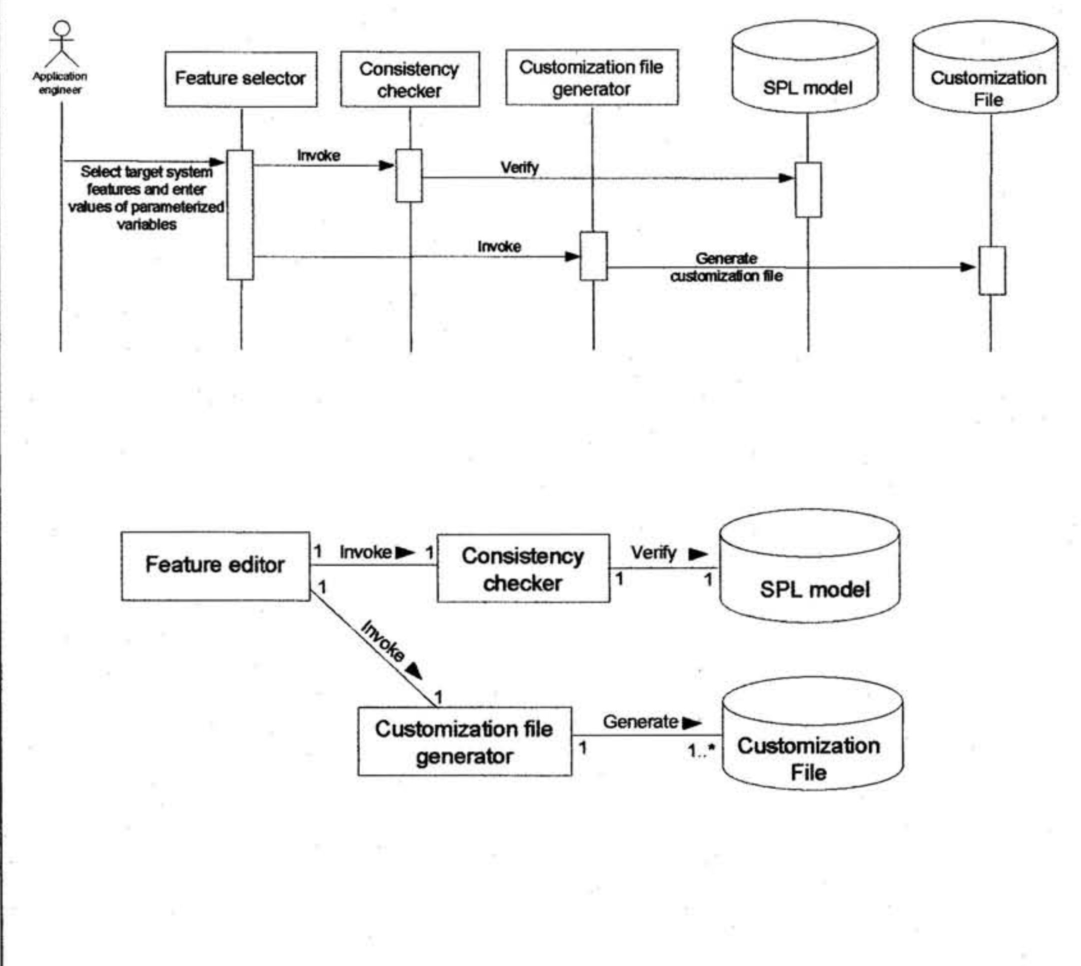
(DCAC pattern – Continue)

- SPL model database: Contains feature tree, feature relations, analysis model, design model, components, and parameterized variables.
- Customization file: Contains feature name, feature selection status (true/false) and values of parameterized variables.

Dynamics

The following scenario depicts the customization process of a target system:

- Application engineer selects desired features for a target system using feature selector component.
- Consistency checker is invoked to verify selection by consulting the SPL model.
- Generate a customization file, which will be used by the client application for dynamic customization at run time.



(DCAC pattern – Continue)

Step 2: Target application interaction
The Dynamic Client Application Customization (DCAC) Pattern divides an interactive application into three components:

- Customizer component
- User interface component
- Web Service component

Customizer component contains all customization information for a single target system. At run time, the customizer object reads the customization file and stores all customization information in the customizer object's local storage (arrays, data table, etc.) to be used for customizing the client application user interfaces and their workflows. Customization information consists of enabled or disabled features and parameterized variables.

User interface component is responsible for accepting input from users and allowing invocation of possible service requests. It involves the sequencing of web services invocation and handling of message communication based on the customizable workflow. It is also responsible for displaying results to users coming from the web service component.

Web Service component is a collection of functional methods that are packaged as a single unit and published in the Internet, Intranet, or Extranet in a private or public UDDI for use by other software programs, in this case the user interface component.

(DCAC pattern – Continue)

Class Customizer	Collaboration
Responsibility - Reads customization information from the customization file / database	- Customization file

Class Web service	Collaboration
Responsibility - Process a service request based on provided input - Returns results of processed requests	- User interface

Class User interface	Collaboration

Responsibility
- Calls customizer class to:
  - Enable or disable user interface components based on selected features
  - Customize user interface
  - Customize workflow by setting up appropriate method calls and calls to other user interfaces based on selected features
- Invoke and pass parameters to appropriate web service(s)
- Receives results from web service(s)
- Display information to the user
	- Customizer
	- Web service

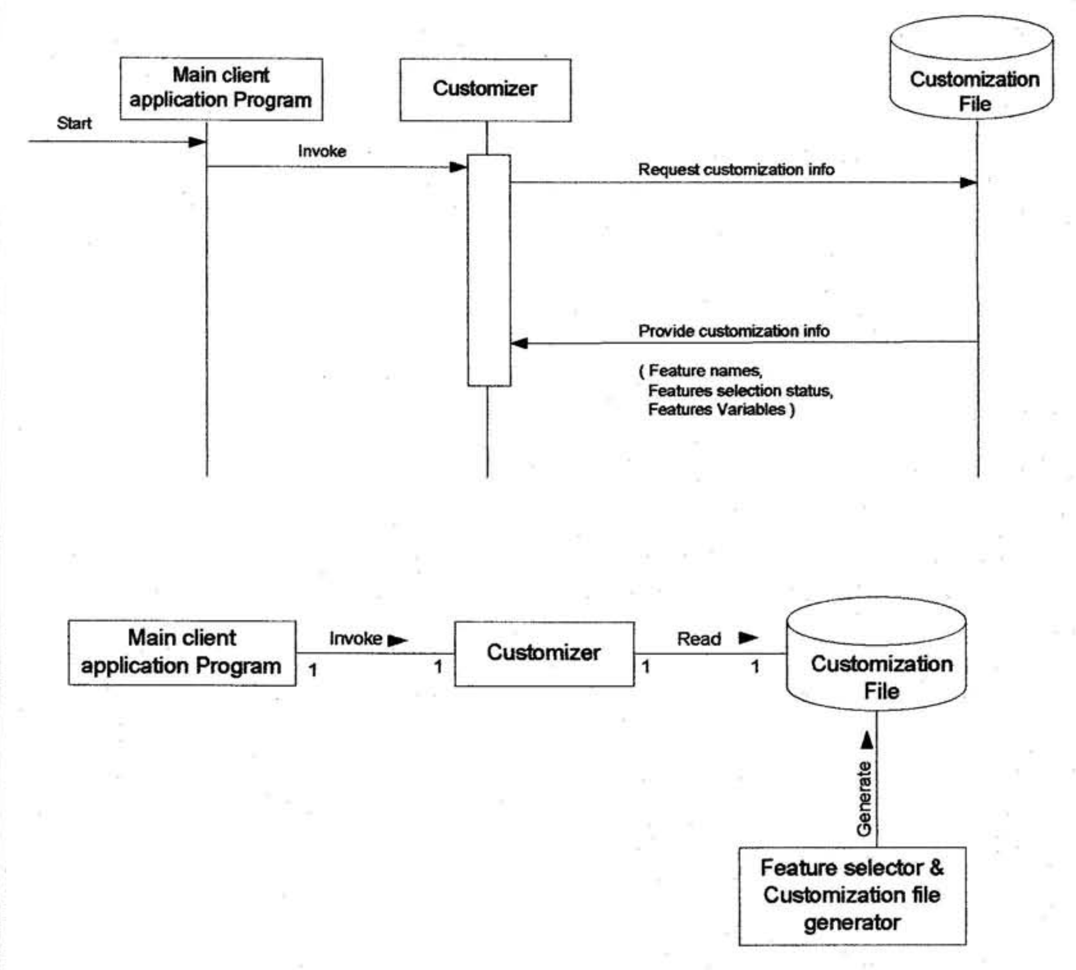
(DCAC pattern – Continue)

Dynamics

Once the target application features are selected in the SPL customization step, the application will be ready for execution. The application interaction step describes the two processes that occur at execution time: dynamic customization and object interactions.

Step 2-1: Shows how the client application is dynamically customized at run time.

- Starts main client application program.
- Customizer object is invoked at main client application program startup.
- Customizer object reads customization information once from the customization file that is generated by the customization file generator.
- Customization information can be read by all user interface objects through the customizer object.



(DCAC pattern – Continue)

Step 2-2: Shows how user interface objects interact with service requests using the DCAC pattern:

Customization of user interface at run time

- User invokes a user interface.
- User interface requests customization information from customizer object.
- User interface reads the customization information to:
    - Customize user interface components
    - Defining appropriate calls to web services based on selected features.
    - Define appropriate calls to other user interface objects.
    - Update parameterized variables.

  Customization is based on feature selection information stored in the customization file.

User interface and web service interaction

- User requests an activity by entering input data and clicking a button.
- User interface object passes the activity request and input data to a web service method(s).
- Web service processes the request and passes the results to the user interface object. A web service may also request services from other web services.
- User interface object displays results received from web service.

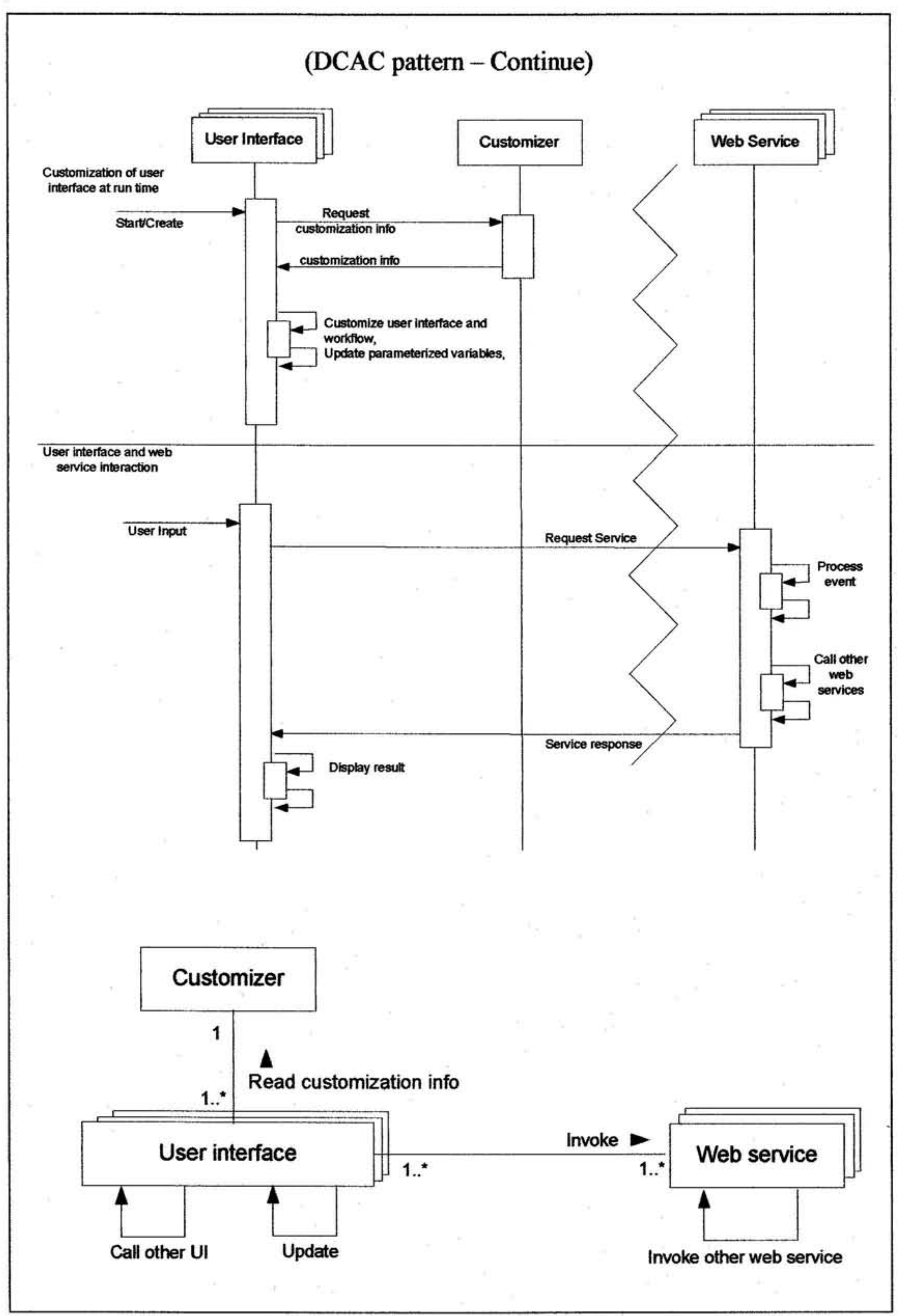


Figure 5-2 Dynamic Customization Workflows (DCAC) Pattern

5.2.1 Development of DCAC pattern

This section describes the development of the DCAC pattern. Two examples from the hotel software product line will be presented to illustrate this development:

- Main Reservation User Interface.
- Reserve Room User Interface.

The first example (see section 4.2.5) shows how alternatives and optional features are treated in the source code, while the second example shows how a service request is performed using web services. Both examples will explain the transition of design into implementation.

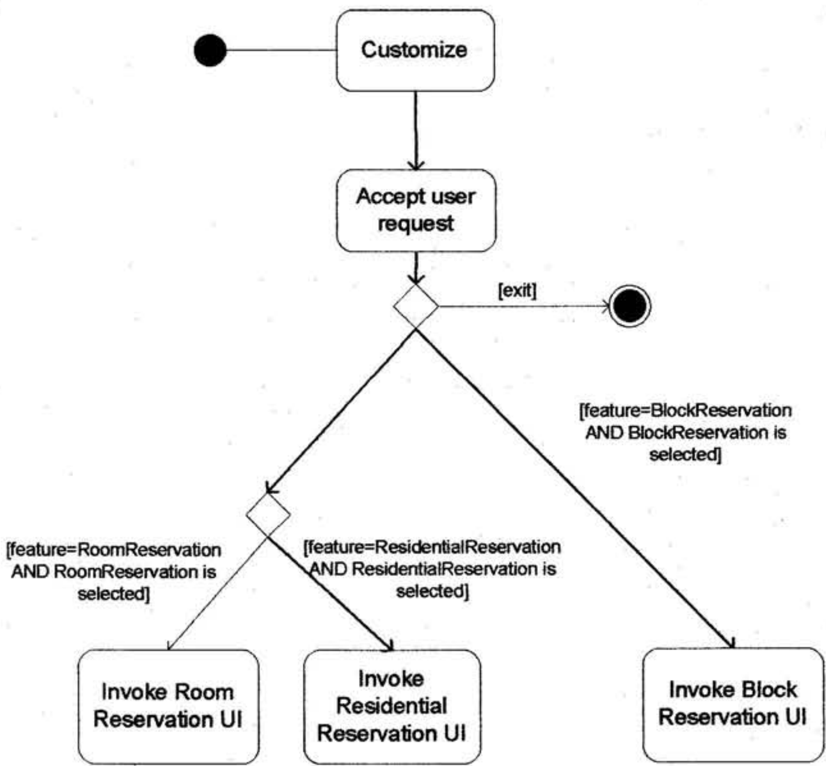


Figure 5-3 Activity Diagram - Main Reservation UI

Figure 5-3 shows a customizable activity diagram for the "MainReservation" user interface. This diagram shows "ResidentialReservation" UI and "RoomReservation" UI as mutually exclusive alternatives where only one of them can be invoked by clicking the single reservation button of "MainReservation" user interface (Figure 5-4). "BlockReservation" UI, on the other hand, belongs to an optional feature. It will be either enabled or disabled based on whether the BlockReservation feature is selected by the user.

The customizable SPL application uses the customization file generated in the application engineering phase to customize a target system at run time (step 1 of DCAC pattern). The customizer object reads the customization file once and stores all customization information in the customizer object's local storage (arrays, data table, etc.) to be used for customizing the client application user interfaces and their workflows. The MainReservation UI is customized by reading the feature selection and the value of parameterized variables from the customizer object to enable or disable buttons and set appropriate display variables. Its workflow is customized by setting features to true or false and applying feature condition settings to user interface calls and web service invocations (step 2 of DCAC pattern). The following explains the customization in more detail.

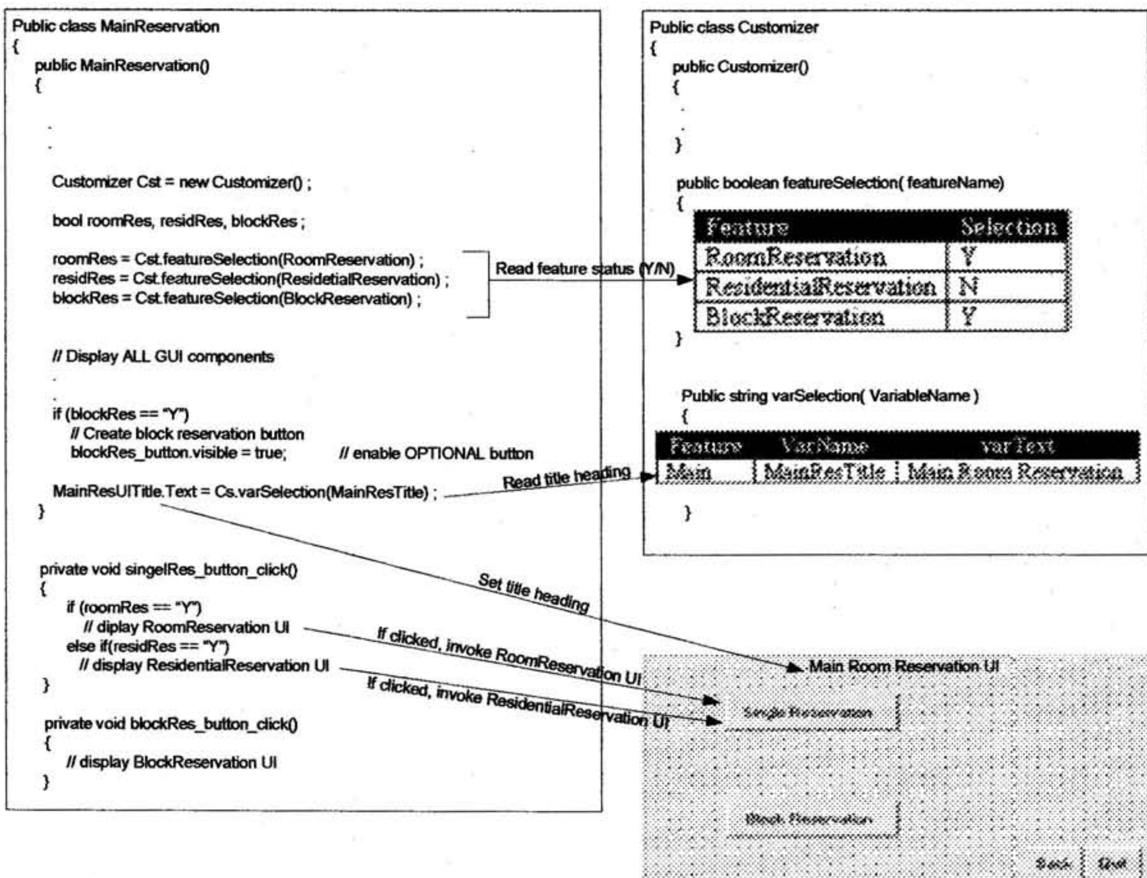


Figure 5-4 Customization phase - Main Reservation UI

Figure 5-4 shows an actual implementation of the activity diagram in Figure 5-3. It shows how the MainReservation UI object can be customized at run time and how it interacts after the dynamic customization.

Customization of client application at run time:

- Object MainReservation is customized by reading the feature selections stored in the customizer object and stores them in local variables, where they will be used throughout the MainReservation object. Local feature variables roomRes, residRes, and blockRes store the RoomReservation, ResidentialReservation, and BlockReservation feature decisions respectively and are set to "Y" or "N", depending on whether the feature is selected or not.

- During the customization process, optional button "Block Reservation" is created if *blockRes* is equal to "Y" and ignored otherwise.

```
if (blockRes == "Y")
    // Create block reservation button
    blockRes_button.visible = true;
```

- During the customization process, the parameterized variable MainResTitle is read from the customizer object to set the appropriate header title of the "MainReservation" user interface.

```
MainResUITitle.Text = Cst.varSelection(MainResTitle);
```

User interface object interaction:

After the dynamic customization process is complete, the MainReservation user interface is ready to accept user input.

- If Single Reservation button is invoked, either "ResidentialReservation" UI or "RoomReservation" UI will be called, depending on whether RoomReservation feature or ResidentialReservation feature is selected.

```
if (roomRes == "Y")
  // display RoomReservation UI
else if(residRes == "Y")
  // display ResidentialReservation UI
```

- If Block Reservation button is enabled and invoked, "BlockReservation" UI will be called.

```
private void blockRes_button_click()
{
        // display BlockReservation UI
}
```

- Since "MainReservation" UI has no service request to process, there will be no web services involved at this user interface.

The second example shows how a service request is processed in the "RoomReservation" UI. Once the "RoomReservation" UI is invoked, it initiates the dynamic customization process, described in the previous example. The user interface is now ready to accept user input and service requests. For the illustration, make single reservation service request is explored.

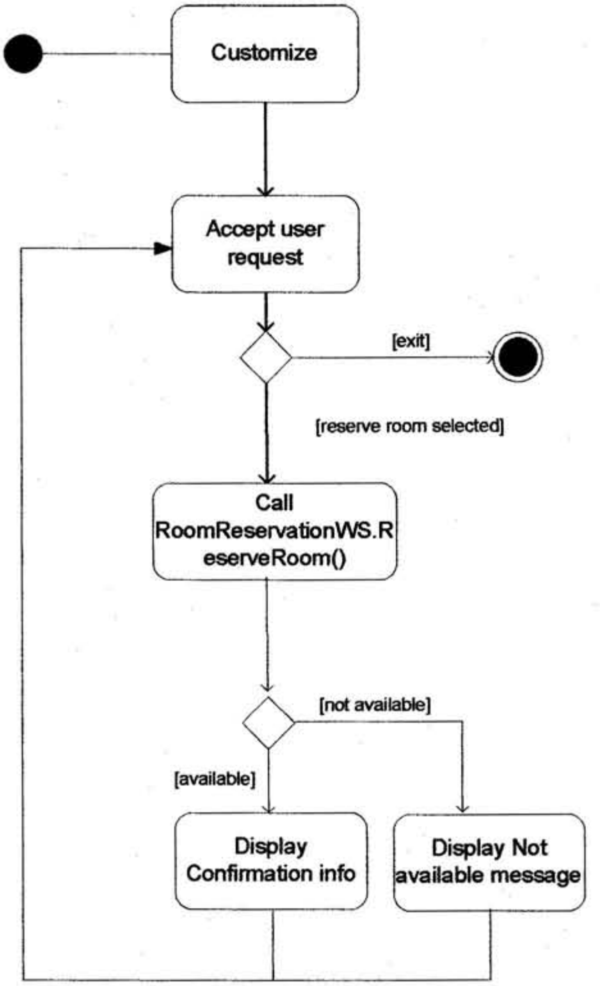


Figure 5-5 Activity Diagram – RoomReservation UI

Figure 5-5 is an activity diagram showing the workflow of processing a single reservation (Reserve button clicked). It has the following activities:

- Customize "RoomReservation" UI
- Accept user input that is required to make a single reservation, such as name, address, duration, and credit card, etc.
- Accept user request to process a single reservation.

- RoomReservationWS will be invoked. Web service method ReserveRoom() will process the request.
- Web service method ReserveRoom() will invoke AvailabilityWS web service and call SetSingleAvailability() method. This method will attempt to update the room availability list in the database.
- Results will be returned to "RoomReservation" UI.
- A confirmation or a decline message will be displayed in the "RoomReservation" UI.

Figure 5-6 is a collaboration diagram for making a single reservation (reserve button clicked). It shows all required objects and their interaction. Since "RoomReservation" UI object has no decision related to alternative or optional feature selection, the customizer object is not shown in the collaboration diagram. Making single room reservation collaboration diagram is implemented in figure 5-7.

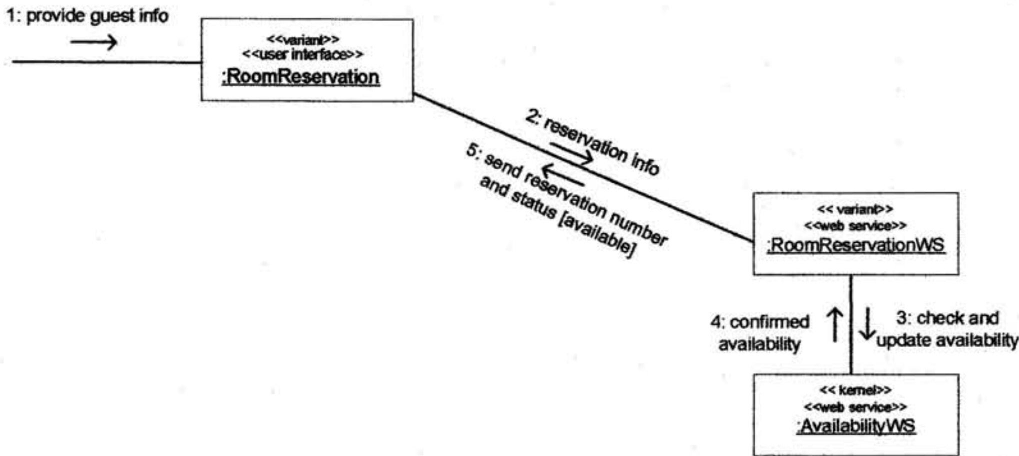


Figure 5-6 Collaboration Diagram – RoomReservation

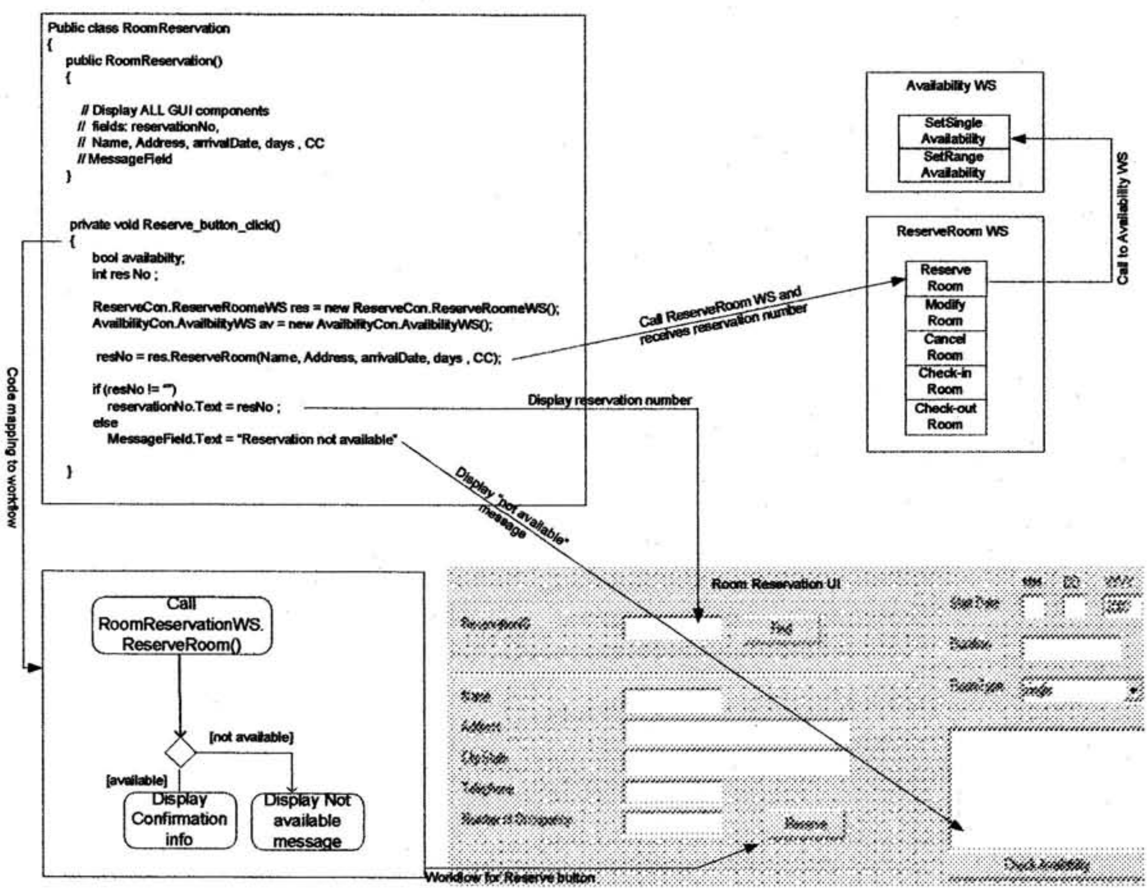


Figure 5-7 Implementation - RoomReservation UI

Figure 5-7 is an implementation sample of the "RoomReservation" UI object of the collaboration diagram in Figure 5-6. The following will explain the process:

- RoomReservation user interface object is responsible for all communication with RoomReservationWS methods. It passes input parameters entered in the graphical user interface and calls ReserveRoomWS web service invoking ReserveRoom() method.

  ReserveCon.ReserveRoomWS res = new ReserveCon.ReserveRoomeWS();
  resNo = res.ReserveRoom(Name, Address, arrivalDate, days , CC);

- ReserveRoom() web service method of the ReserveRoomWS web service processes the entire service request. A web service may call one or more web services methods. In this case, SetSingleAvailability() method is invoked from the AvailabilityWS web service.
- ReserveRoom() method returns a numeric reservation number, which is stored in the local variable *resNO* of the RoomReservation user interface and the database.
- RoomReservation user interface displays results received.
- Either reservation number or a decline message is displayed to the user.

5.2.2 Advantages of DCAC approach:

- Client application is involved with workflows only. All service requests are processed using web services. This makes it easier to develop a client application quickly and construct it for dynamic customization at run time. Also, this makes it easier to develop web services individually and incorporate them in the SPL Service-Oriented application.
- No source code extraction or update of source code to fit target systems. The customizable SPL application is created once with all possible features incorporated in the interactive application. Target systems are dynamically customized at run time by reading the customization file that is generated during the application engineering phase using the feature selector component.
- No recompilation of target system applications. The customizable SPL application is compiled only once to generate an executable SPL application that can be customized dynamically at run time.
- Since Web Services process all service requests, it is easier to test each request separately using the provided standard interface of web services.
- Software reuse: Web services are developed once and can be reused by target applications.

5.2.3 Disadvantages of DCAC approach:

- Source code overhead: All optional and alternative source code is interwoven with kernel source code in the interactive client application. Selected optional and alternative source code blocks are invoked at run time using a customization file that includes feature selection.
- No fixed workflows. Workflows are driven by feature selection, which adds an extra activity to the interactive application by applying decisions at run time to which optional or alternative web service to call, or what UI components to display.
- There is no separation of concerns between kernel source code and optional and alternative source code.

5.3 Introduction to the customization approaches based on separation of concerns

The second and third development approaches apply the Aspect-Oriented programming (AOP) and framing technology (FT) concepts, described in Chapter 2, to separate the optional and alternative source code from kernel source code, which is referred to as separation of concerns. The concept of AOP and FT will be tailored to accommodate the capturing of variability based on feature grouping of related optional and alternative source code in a variable source code file that is used for the purpose of customization and integration with consistency checking support.

The second and third development approaches apply the AOP concept of separation of concerns and code weaving. Separation of concerns is used to separate variable source code form kernel source code in a variable source code file. In AOP for product lines [Leasint04, Loughran04a, Anastasopoulos04], the aspect file is used to store all required source code needed for a specific target application with no consideration to variable source code. Therefore, for every target application an aspect file has to be manually created to satisfy its requirements. In this research, the variable source code file stores all variable source code to be integrated (weaved) according to the two proposed automatic customization and integration processes with no manual modification to the variable source code file when deriving a target application. A detailed comparison of AOP and this research was described earlier in section 3.8.2.

The next two development approaches apply the concept of FT by grouping feature related variable source code in different files, similar to frames, which are composed into one variable source code file. But unlike frames, the two development approaches in this research provide a systematic method for relating source code to features for the purpose of automating customization and integration of SPL applications with consistency checking support to verify feature selection. A detailed comparison of FT and this research was described earlier in section 3.8.2.

The second development approach applies separation of concerns for applications that are customized dynamically at run time. Optional and alternative source code is separated from kernel source code in a variable source code file. Separated source code is then integrated with kernel source code in the code weaving process to generate the complete SPL application source code. Integrated source code is compiled once and becomes ready for execution. Target applications are then customized using the feature selection and consistency checking process to generate a customization file that stores feature selection and parameterized variables. The executable SPL application reads this file to apply customization of client applications at run time.

The third development approach applies separation of concerns for applications that are customized during source code integration time. Optional and alternative source code is separated from kernel source code in a variable source code file. But unlike the second development approach where feature selection is performed after source code integration

is complete, the feature selection process in the third development approach is performed before the source code is integrated. The code weaver engine reads feature selection and integrates only selected optional and alternative feature source code with kernel source code. The integrated source code is compiled and becomes a customized executable target application.

5.4 Development of dynamic customization of client application with separation of concerns

The second development approach is an extension to the first method (DCAC), described in sections 5.2 to include the separation of concerns. It is based on the dynamic customization of client applications, where objects are customized at system run time. However, this method provides the separation of optional and alternative source code from kernel source code into a variable source code file. Figure 5-8 shows the overall method of DCAC-SC from the SPL engineering phase to the application engineering phase (SPL customization), which is the same as the DCAC method. It also depicts the separation of concerns that is added to the DCAC method. It shows the needed facilities to create the separation of concerns, feature selection, consistency checking, and integration of kernel source code with optional and alternative source code. The result of the integration process is a combined set of source code for the entire software product line including all optional and alternative source code. The source code integration process and compilation are performed only once to generate an executable SPL system. Target systems will rely on the dynamic client application customization at system run time, which is identical to that produced by the first approach (DCAC).

Similar to DCAC approach, the customizable SPL system uses the generated customization file produced in the application engineering phase to customize a target system at run time. The customizer object reads the generated customization file and stores all customization information in the customizer object's local storage (arrays, data table, etc.) to be used for customizing the client application user interfaces and their workflows. User interfaces are customized by enabling or disabling buttons, and by setting appropriate display variables. Workflows are customized by customizing decisions to which user interface to call or which web service to invoke.

Since this approach is based on SPL Service-Oriented Architecture, separation of concerns focuses on:

- Separation of optional and alternative service calls.
- Separation of optional and alternative calls to user interfaces.
- Separation of optional and alternative user interface components, such as buttons headings, and images.

This approach is described in the Dynamic Client Application Customization with Separation of Concerns (DCAC-SC) Pattern in Figure 5-9.

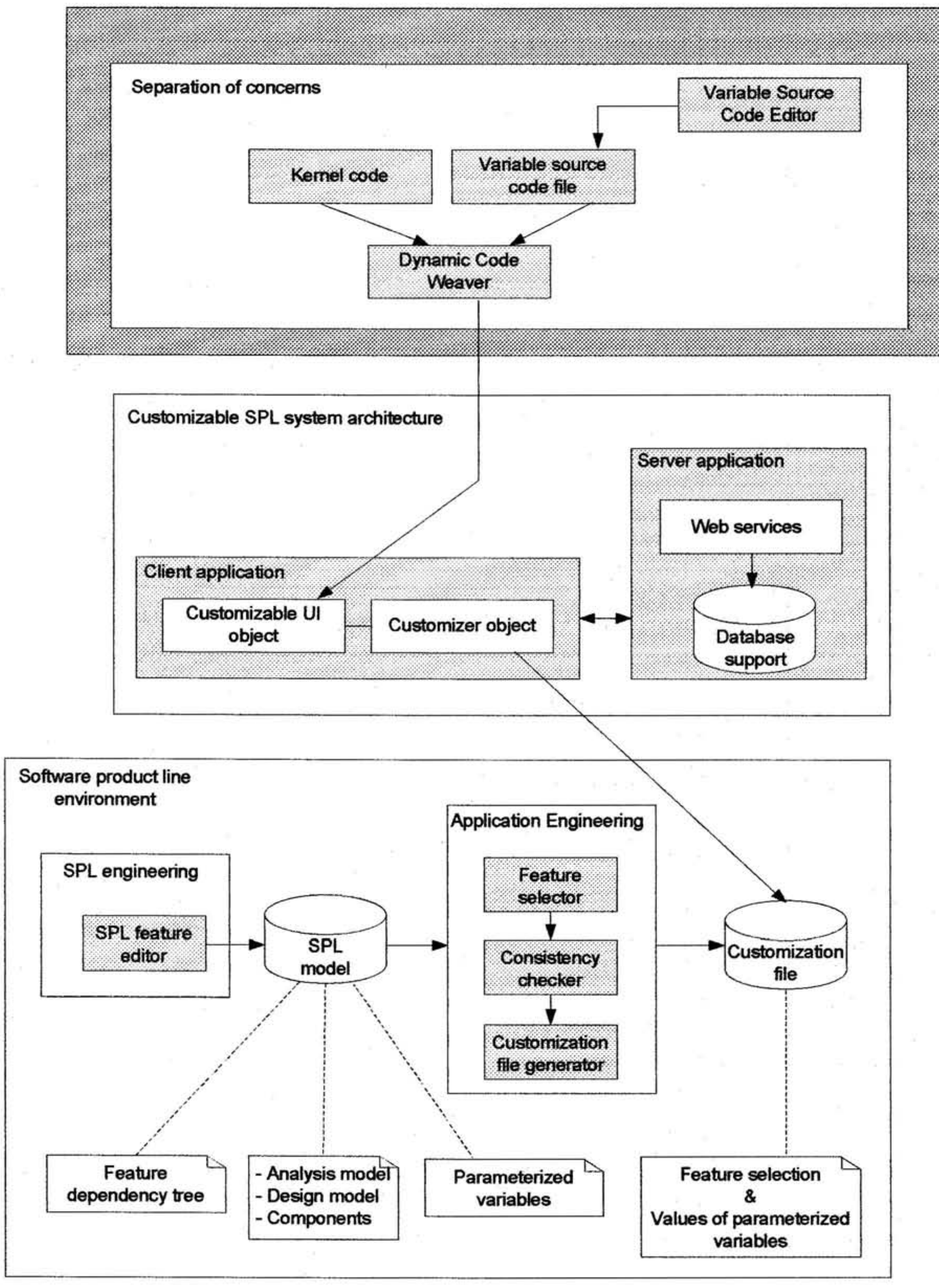


Figure 5-8 Conceptual overview of DCAC-SC approach

Dynamic Client Application Customization with Separation of Concerns Pattern

Intent
Provide a consistent reusable solution to the implementation architecture of a software product line using web services with provision for dynamic client application customization and separation concerns.

Motivation
This pattern is an extension to the DCAC pattern, which does not address the issue of separation of concerns. This issue needs to be introduced for the purpose of reducing complexity of developing SPL applications, maintenance, and system evolution.

Solution
The idea behind the (DCAC-SC) pattern is the development of dynamic client application that can be customized at system run time by separation of concerns between kernel source code and optional and alternative source code.

The DCAC-SC Pattern has four main steps:

1. Separation of concerns between kernel and variable source code
2. Code weaving
3. SPL Customization (the same as the DCAC pattern)
4. Target application interaction (the same as the DCAC pattern)

The above steps have to be performed in sequence. First, separation of concerns and code weaving have to be performed. The SPL application can then be customized by selecting desired features. Target applications are compiled to produce an executable SPL application.

Step 1: Separation of concerns between kernel and variable source code:

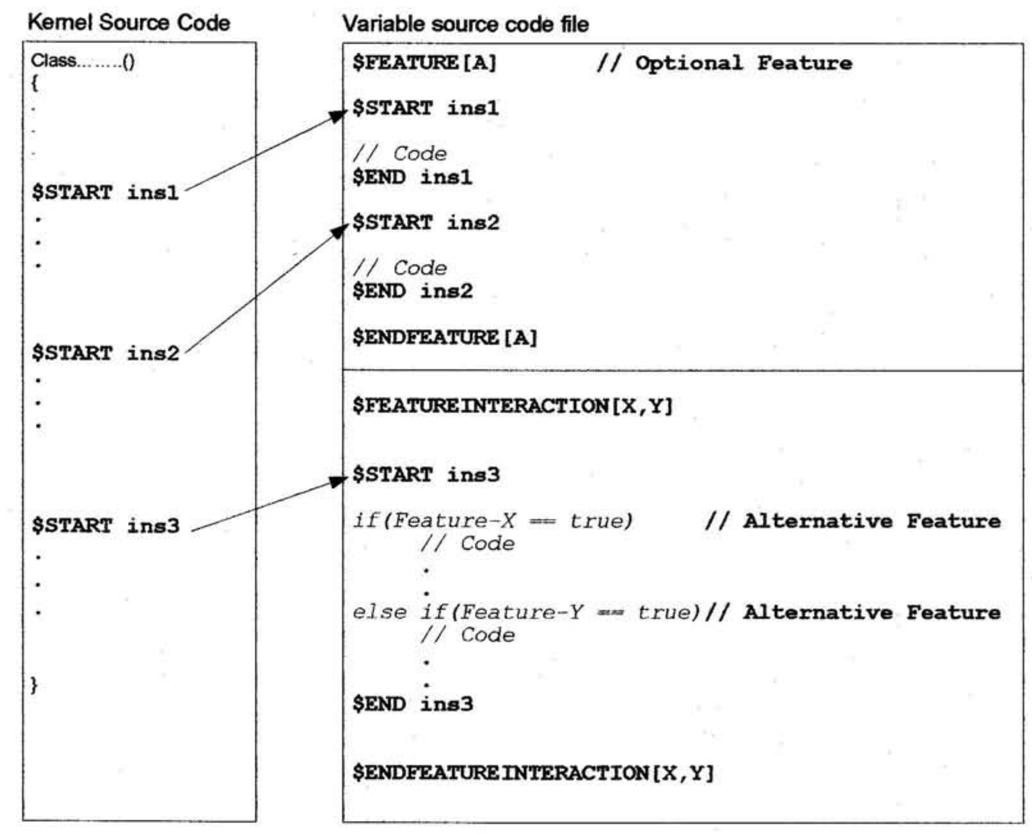
This step involves separating kernel source code from optional and alternative source code into a variable source code file where separated source code is grouped by features. Optional and alternative source code is identified by unique insertion point names in the variable source code file. Insertion points have to be also included in the kernel source code to specify the location where optional and alternative source code will be inserted.

(DCAC-SC pattern – Continue)

Dynamics

The following scenario depicts the dynamic behavior of separation of concerns:

- Create application classes with kernel source code.
- Create a variable source code file that contains source code related to alternative and optional features.
- Add insertion points to kernel source code where optional and alternative source code from the variable source code file will be inserted.



(DCAC-SC pattern – Continue)

Language description:

- Kernel source code
  - $START <<insertion name>>: Specifies insertion location in kernel source code

- Variable source code file
  - $START <<insertion name>>: Identifies optional or alternative source code that needs to be inserted at the location specified in the kernel source code.

  - $END <<insertion name>>: Specifies the end of insertion code.

  - FEATURE [<<feature name>>]: Groups optional and alternative source code in a feature block. Feature blocks are integrated with kernel source code during the code weaving process based on insertion names.

  - FEATUREINTERACTION[<<feature 1, feature 2, …>>]: Groups related features source code that requires decisions on which source code to execute at run time. If-then-else statement is used within the insertion name of the feature interaction block with feature identifiers in the decision statement to be integrated as-is in the kernel source code based on the language used to develop the SPL application. At run time, only one of the decisions will be executed based on feature selection during SPL customization.

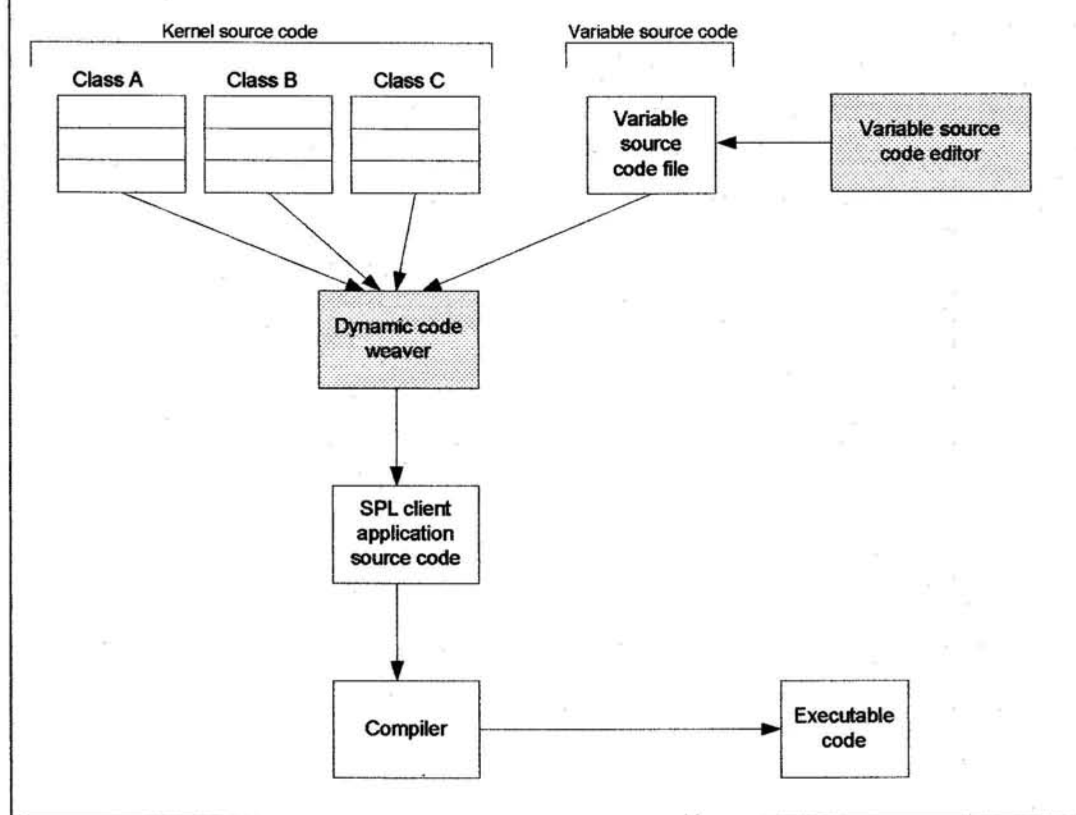
  - ENDFEATUREINTERACTION []: Specifies the end of feature interaction source code.

(DCAC-SC pattern – Continue)

Step 2: Code weaving
This step combines kernel source code with optional and alternative source code from the variable source code file. This process is based on the Code Weaver component, which reads the variable source code file and inserts all source code blocks from that file into the kernel source code at the specified insertion locations.

Dynamics
The following scenario depicts the dynamic behavior of code weaving process:

- Run the code weaver component.
- Read optional and alternative source code from the variable source code file and integrate it into kernel classes at the specified insertion point locations.
- Compile integrated source code to generate an executable dynamic SPL application.



(DCAC-SC pattern – Continue)

The following diagram shows the complete process of separation of concerns and source code integration:

Create Kernel code in classes

Develop client application classes

Kernel source code

Add Insertion points where all code from the feature file will be inserted

Add insertion points in kernel classes

Create a variable source code file that contains code related to alternative and optional features

Create variable source code file

variable source code file

Read variable source code file and integrate all source code into kernel classes at the specified insertion locations

Weave code

Kernel source code

Compile the integrated source code to generate an executable dynamic SPL system

Compile

Integrated source code

Run executable SPL application

Executable code

(DCAC-SC pattern – Continue)

Step 3: SPL Customization
This step is identical to the SPL customization step in the DCAC pattern. It involves selecting desired optional and alternative features to be included in the target application. The feature selector component provides a facility to make feature selection from a SPL model and run consistency checks to verify selections. Once features are selected, selection information will be stored in the customization file using the customization file generator. The dynamic client application is customized by reading the generated customization file at run time. This step is described in full in step 1 of the DCAC pattern.

Step 4: Target application interaction
This step is identical to the target application interaction step in the DCAC pattern. This step follows the SPL customization step. Once the target application features are selected, the application will be ready for execution. This step describes how the client application is customized dynamically at run time, and how user interface objects interact with service requests. This step is described in full in step 2 of the DCAC pattern.

Figure 5-9 Dynamic Client Application Customization with Separation of Concerns Pattern

5.4.1 Development of DCAC-SC pattern

This section describes the implementation of the DCAC-SC pattern. The "MainReservation" UI from the hotel product line will be used for this illustration. The ideas presented in this example can be applied to all user interfaces. The example will illustrate the approach of separating optional and alternative source code from kernel source code into a variable source code file. It will also explain the integration of variable source code with kernel code to produce a dynamically customizable SPL application. The separation and integration of source code is based on the DCAC-SC pattern.

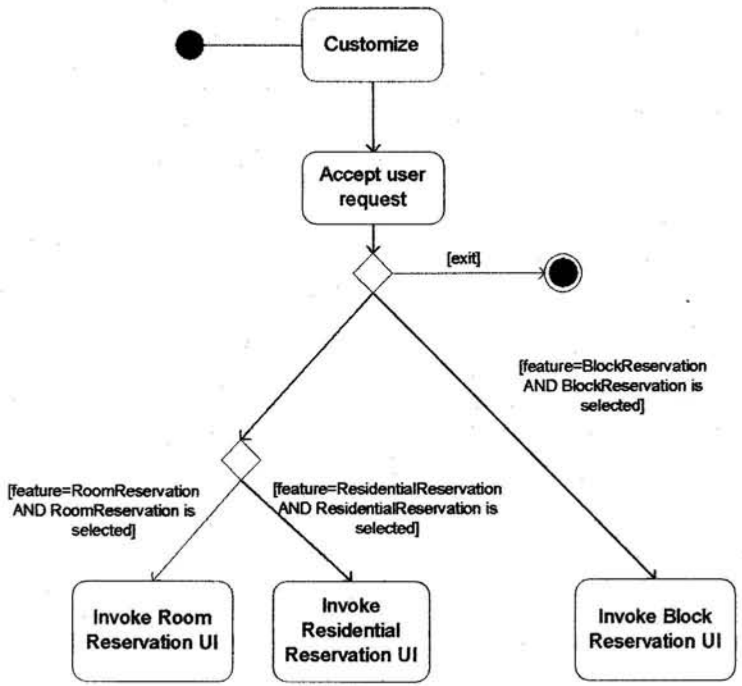


Figure 5-10 Activity Diagram - Main Reservation UI

Figure 5-10 shows the same customizable activity diagram used to illustrate the implementation of method 1 (DCAC). However, this example includes separation of concerns, which is not addressed in the previous method. The customizable activity diagram represents the "MainReservation" user interface. It shows "ResidentialReservation" UI and "RoomReservation" UI as mutually exclusive alternatives where only one of them can be invoked by clicking the single reservation button of "MainReservation" user interface (Figure 5-11). "BlockReservation" UI, on the other hand, belongs to an optional feature. It will be either enabled or disabled based on whether BlockReservation feature is selected by the user. These decisions are set during application run time, the same as for the DCAC pattern.

All source code in the variable source code file will be extracted and integrated with the kernel source code. The result of the integration process is a SPL system that is identical to method 1 (DCAC).

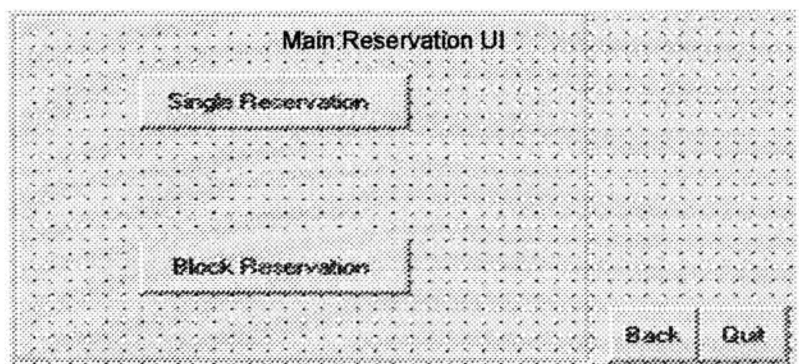


Figure 5-11 MainReservation graphical user interface

Figure 5-11 is the graphical user interface for "MainReservation" UI class. It shows two event buttons: Single reservation and Block Reservation. If single reservation button is clicked, either "RoomReservation" UI or "ResidentialReservation" UI will be invoked. Also BlockReservation button will either be visible or invisible based on feature selection during the SPL customization process.

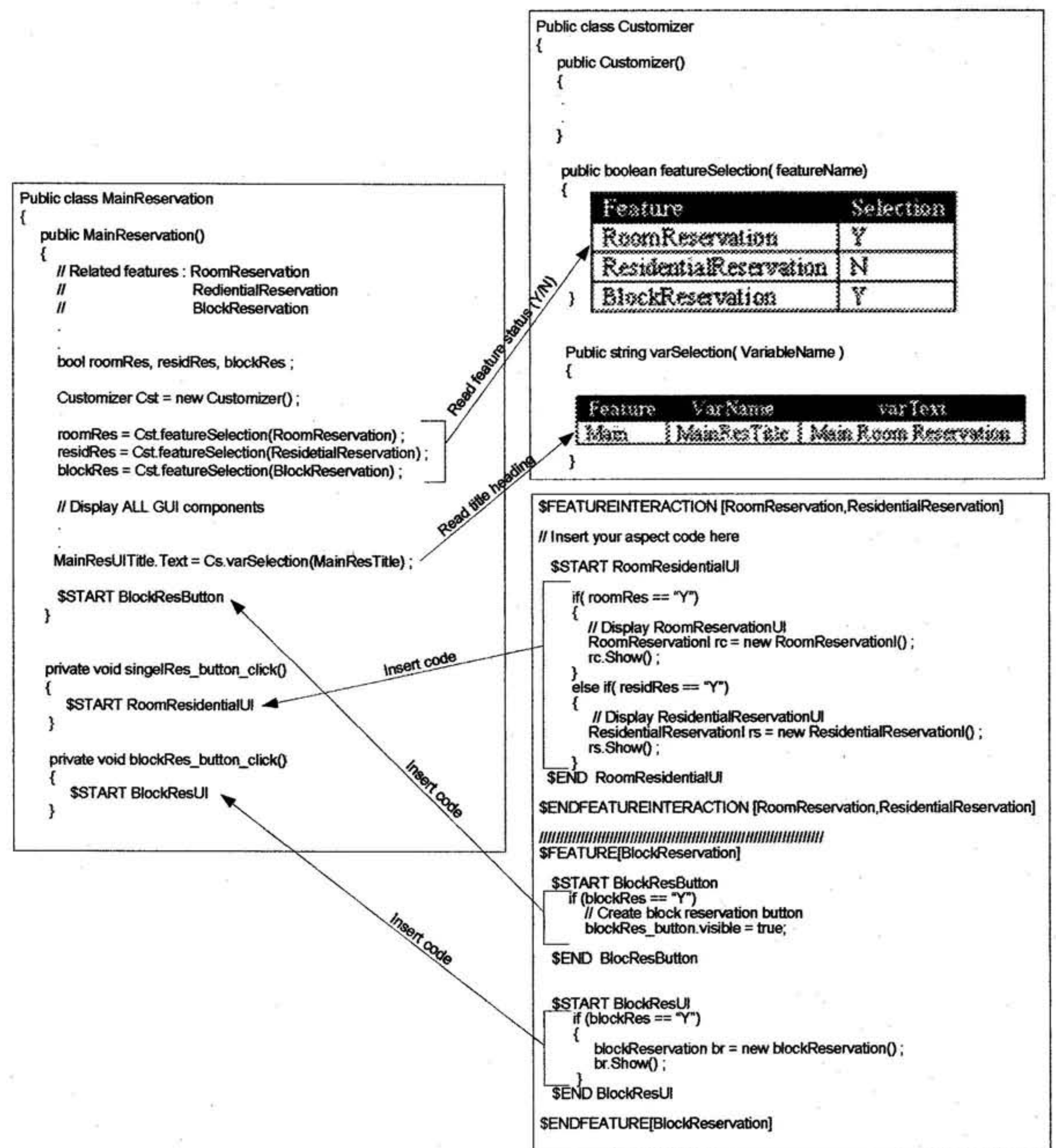


Figure 5-12 Implementation - Main Reservation UI

Figure 5-12 shows both the kernel source code of the "MainReservation" UI and variable source code in the variable source code file. Insertion points are the key for integrating

variable source code with kernel source code. The code weaver engine is responsible for this integration. The code weaver reads all application class files and locates insertion points. It then reads the variable source code file and adds variable source code at the location of the insertion points based on matched feature names. The key command $START is followed by an insertion name, which is used for the integration process. Both kernel source code and variable source code in the variable source code file contain the same insertion point name. Insertion point in the kernel source code identifies the location of the insertion, and insertion name in the variable source code file identifies which variable source code is to be inserted.

Based on the DCAC-SC approach, all optional and alternative feature source code in the variable source code file is inserted in the kernel source code at the location of the insertion point; customization is done at run time. For example, the insertion point *$START BlockResButton* refers to the optional feature "BlockReservation" in the variable source code file. The variable source code will be inserted in the kernel "MainReservation" UI class at the place of the insertion point: *$START BlockResButton*. At run time, this button will be either visible or invisible based on feature selection.

```
Public class MainReservation
{
   public MainReservation()
   {
      // Related features : RoomReservation
      //                    RedientialReservation
      //                    BlockReservation
      .
      .
      // Display ALL GUI components
      .
      .
      Customizer Cst = new Customizer() ;

      bool roomRes, residRes, blockRes ;

      roomRes = Cst.featureSelection(RoomReservation) ;
      residRes = Cst.featureSelection(ResidetialReservation) ;
      blockRes = Cst.featureSelection(BlockReservation) ;

      // START BlockResButton
      if (blockRes == "Y")
        // Create block reservation button
        blockRes_button.visible = false;

      MainResUITitle.Text = Cs.varSelection(MainResTitle) ;
   }


   private void singelRes_button_click()
   {
      // $START RoomResidentialUI
      if( roomRes == "Y")
      {
         // Display RoomReservationUI
         RoomReservationI rc = new RoomReservationI() ;
         rc.Show() ;
      }
      else if( residRes == "Y")
      {
         // Display ResidentialReservationUI
         ResidentialReservationI rs = new ResidentialReservationI() ;
         rs.Show() ;
      }
   }

   private void blockRes_button_click()
   {
         // $START BlockResUI
         if (blockRes == "Y")
         {
            blockReservation br = new blockReservation() ;
            br.Show() ;
         }
   }
```


Figure 5-13 Implementation - Main Reservation UI

Figure 5-13 shows the "MainReservation" UI class after the integration process. In this class, the BlockReservation feature and the RoomReservation feature are inserted in the kernel source code. Inserted blocks are:

- Insertion point $START BlockResButton in the kernel source code is replaced with the following source code from the variable source code file:

```
// START BlockResButton
if (blockRes == "Y")
    // Create block reservation button
    blockRes_button.visible = true;
```

- Insertion point $START RoomResidentialUI in the kernel source code is replaced with the following source code from the variable source code file:

```
// START RoomResidentialUI
if( roomRes == "Y")
{
    // Display RoomReservationUI
    RoomReservationI rc = new RoomReservationI() ;
    rc.Show() ;
}
else if( residRes == "Y")
{
    // Display ResidentialReservationUI
    ResidentialReservationI rs = new ResidentialReservationI() ;
    rs.Show() ;
}
```

- Insertion point $START BlockResUI in the kernel source code is replaced with the following source code from the variable source code file:

```
// START BlockResUI
if (blockRes == "Y")
{
    blockReservation br = new blockReservation() ;
    br.Show() ;
}
```

Figure 5-14 shows the insertion points needed in the "Mainreservation" UI class. This figure has to be added to the modeling of SPL Service-Oriented Architecture, discussed in chapter 4. Insertion points are depicted from the activity diagram, where feature conditions are stated to show the possible workflows for a single target application. The activity diagram in Figure 5-10 shows three different decisions that need to be set when customizing the target system workflow. These decisions are presented as feature conditions, which are:

- RoomReservation alternative feature condition
- ResidentialReservation alternative feature condition
- BlockReservation optional feature condition

Feature Name	Feature Type	Class Name	Insertion Point Name
BlockReservation	Optional	MainReservation	BlockResButton BlockResUI
RoomReservation	Alternative	MainReservation	RoomResidentialUI
ResidentialReservation	Alternative	MainReservation	RoomResidentialUI

Figure 5-14 MainReservation UI - Insertion points list

5.4.2 Advantages and Disadvantages of DCAC-SC approach:

Since this development approach is an extension to the DCAC, the advantages and disadvantages are the same as of the DCAC approach in sections 5.2.2 and 5.2.3 except for the issue related to separation of concerns. The DCAC-SC method provides an advantage over DCAC by supporting the concept of separation of concerns between kernel source code and variable source code for the purpose of reducing complexity of developing and maintaining software product lines. This issue was one of the disadvantages of DCAC pattern that is solved by the DCAC-SC pattern.

5.5 Development of static customization of client application (SCAC) with separation of concerns

The third development approach also includes the separation of optional and alternative source code from kernel source code. However, it is based on *static* customization of client applications, where objects are customized at source code integration time using a variable source code file, customization file, and an integration engine. Objects are customized by integrating kernel source code with *only* selected optional and alternative source code from the generated variable source code file producing the required source code needed for running a single target application.

In the SCAC method, optional and alternative source code is separated from kernel source code in a variable source code file for the purpose of generating customized target applications. But unlike the DCAC-SC method, where feature selection is performed after source code integration is complete, the feature selection process (SPL customization) in the SCAC method has to be performed before the source code is integrated. The code weaver engine reads feature selection and integrates only selected optional and alternative feature source code with kernel source code. The result of the integration process is an integrated source code file for the customized target application. The SPL customization, source code integration, and compilation are performed for each target application, unlike the DCAC-SC where source code integration and compilation are done only once to generate a dynamically customizable system at run time. This method produces a target application that is already customized and ready to execute.

Figure 5-15 shows a conceptual overview of the approach. It depicts:

- Separation of concerns
- SPL system architecture
- SPL environment

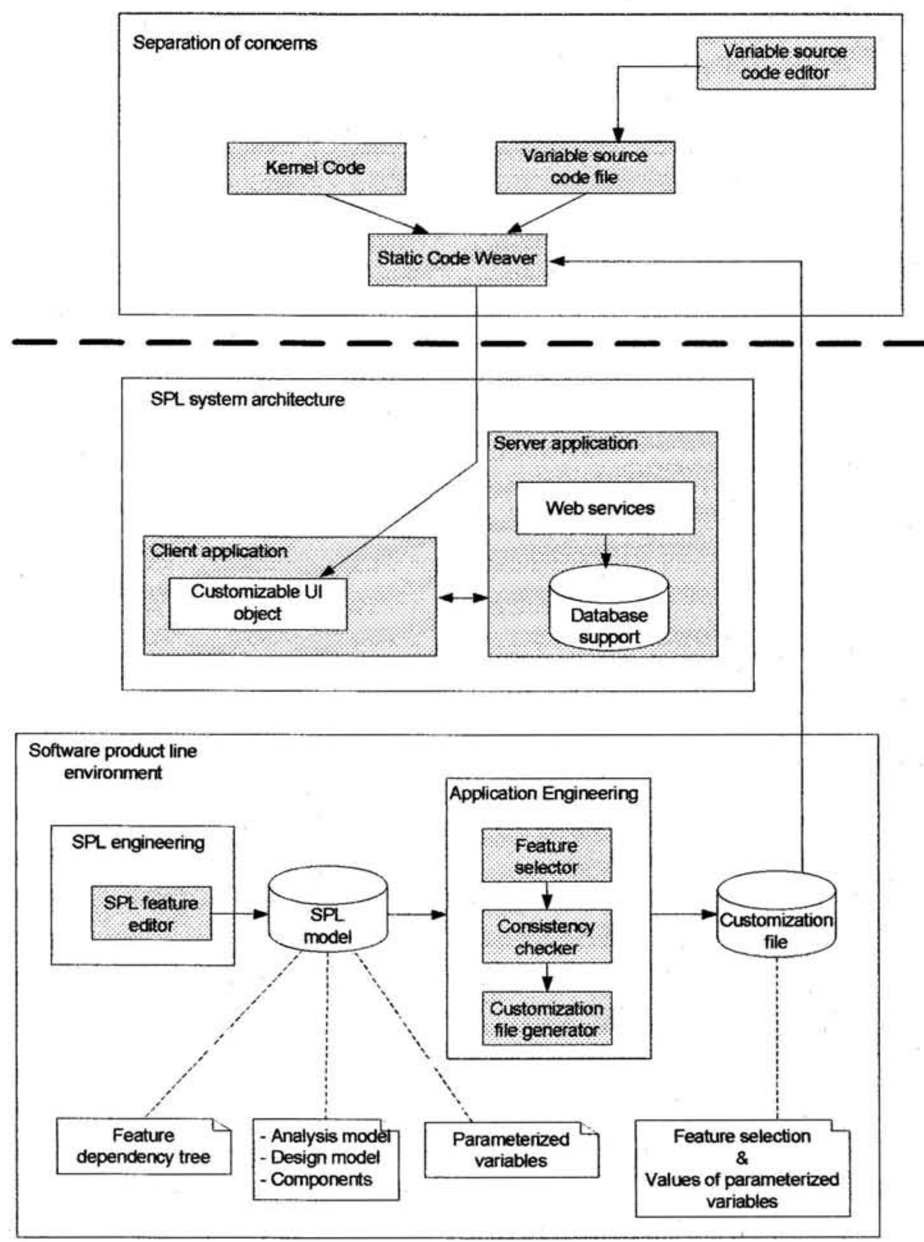


Figure 5-15 Conceptual overview of SCAC approach

Separation of concerns in Figure 5-15 is based on separating kernel source code from optional and alternative source code into a variable source code file where separated source code is grouped by features. The code weaver component is used as an integration engine. The code weaver reads feature selection from the customization file and integrates selected feature source code from the variable source code file with kernel source code.

The SPL system architecture in Figure 5-15 is based on the client/server design pattern, where the client application contains only user interface objects, and the server application contains all web services and database support. In this approach there is no customizer object used, which was required in the DCAC and DCAC-SC for dynamic customization at run time.

The software product line environment in Figure 5-15 shows a conceptual overview of the approach from the SPL engineering phase to the application engineering phase (SPL customization), which is the same as the first and second development approaches (DCAC and DCAC-SC). It shows the needed facilities to create the SPL environment, select target application features, apply consistency checks, and generate a customization file.

This approach is described in the Static Client Application Customization (SCAC) Pattern in Figure 5-16.

Static Client Application Customization Pattern

Intent

Provide a consistent reusable solution to the implementation architecture of a software product line using web services with provision for *static* customization of client application using the concept of separation of concerns.

Motivation

The goal of developing software product lines is to promote flexible software reuse. With the introduction of web services to SPLs, there is a need for developing a systematic approach that enables developers to implement a customizable overall system that can be customized into many single target systems using a systematic method for extracting the required source code for each target system.

Solution

The idea behind the Static Client Application Customization (SCAC) pattern is the separation of concerns between kernel source code and optional and alternative source code for the purpose of extracting only required source code for running a target system.

The SCAC Pattern has four main processes:

1. Separation of concerns between kernel and variable source code
2. SPL Customization
3. Code weaving
4. Target system interaction

The above steps have to be performed in sequence. Variable source code has to be separated from kernel source code in the separation of concerns step. The SPL customization has to be performed next to select the target application features before integrating variable source code with kernel source code in the code weaving step. The customization file generated in the SPL customization step is required in the integration process. Target applications are compiled to produce an executable target application.

Step 1: Separation of concerns between kernel and variable source code

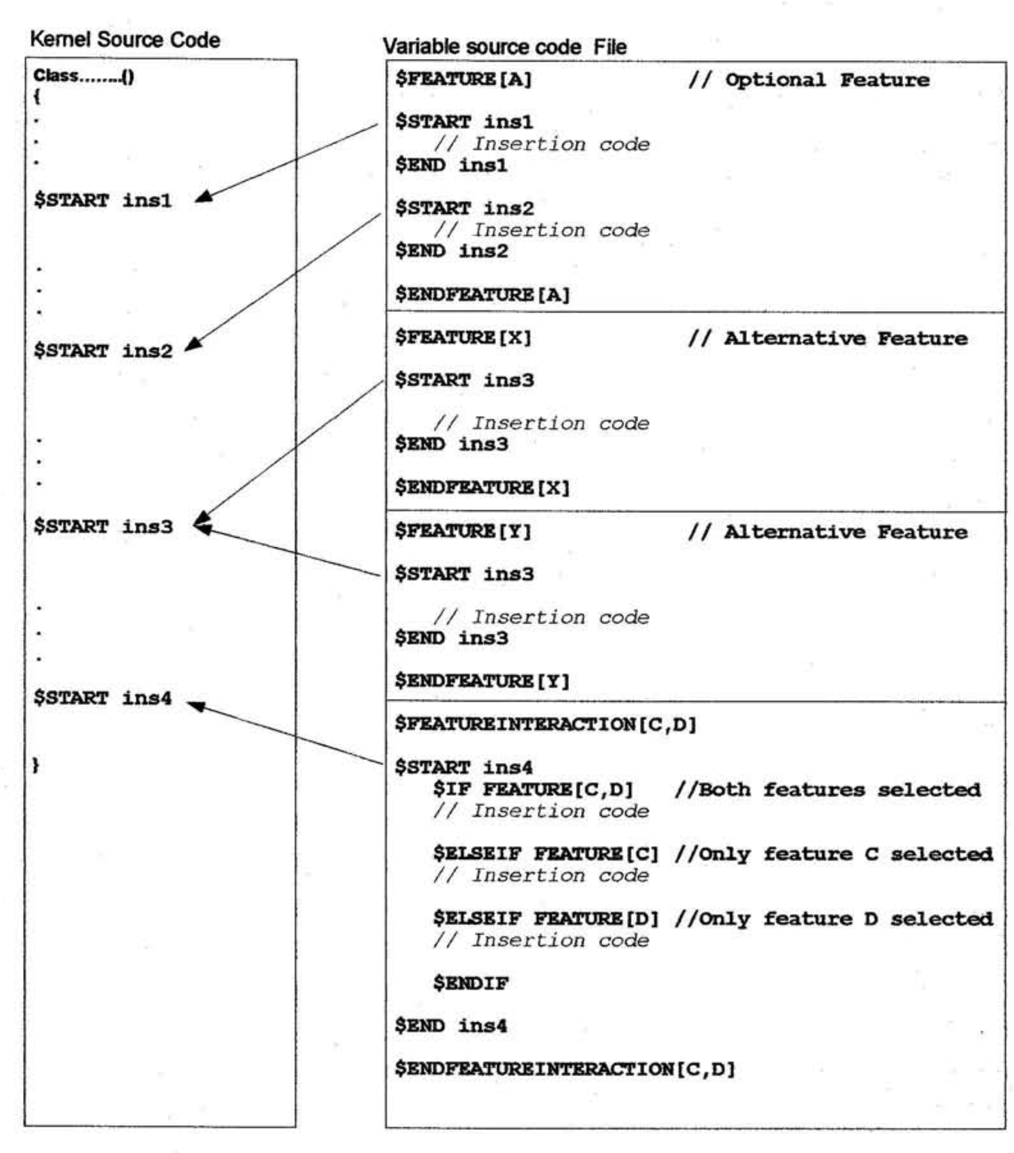
This step involves separating kernel source code from optional and alternative source code into a variable source code file where separated source code is grouped by features. This step is similar to the separation of concerns step in the DCAC-SC pattern, but differs in the construction of the variable source code file to include necessary decisions when more than one feature is involved within an insertion point name. These decisions enable the code weaver engine to integrate only selected variable source code rather than integrating all variable source code as done in the DCAC-SC.

(SCAC Pattern – Continue)

Dynamics

The following scenario depicts the dynamic behavior of Separation of concerns:

- Create application classes with kernel source code.
- Create a variable source code file that contains source code related to alternative and optional features.
- Add necessary decisions within insertion point names for insertions that involve more than one feature (feature interaction).
- Add insertion points to kernel source code where optional and alternative source code from the variable source code file will be inserted, based on feature selection.



(SCAC Pattern – Continue)

Language description:

- Kernel source code
  - $START <<insertion name>>: Used to specify insertion location in kernel source code

- Variable source code file
  - $START <<insertion name>>: Used to identify optional or alternative source code that needs to be inserted at the location specified in the kernel source code.

  - $END <<insertion name>>: Specifies the end of insertion source code.

  - FEATURE [<<feature name>>]: Groups optional or alternative source code in a feature block. Feature blocks are integrated with kernel source code during the code weaving step based on insertion names.

  - FEATUREINTERACTION[<<feature 1, feature 2, …>>]: Groups related feature source code that requires decision on which source code is to be included in the code weaving step.

  - $IF FEATURE [<<feature 1>>, <<feature 2>>, ..]: A programmatic decision point within the FEATUREINTERACTION block that is used to notify the code weaver engine whether to include the following source code block or not based on selected features in the customization file.

  - $ELSEIF FEATURE [<<feature name>>]: A programmatic *ELSEIF* point to be used in case the IF FEATURE statement is false.

  - $ENDIF: Specifies the end of the decision statements.

  - ENDFEATUREINTERACTION []: Specifies the end of feature interaction source code.

(SCAC Pattern – Continue)

Step 2: SPL Customization
This step is identical to the SPL customization step in the DCAC and DCAC-SC patterns. However, this step has to be performed before integrating variable source code with kernel source code in the code weaving step. It involves selecting desired optional and alternative features to be included in the target application. The feature selector component provides a facility to make feature selection from the feature model and run consistency checks to verify feature selections. Once features are selected, selection information will be stored in the customization file by the customization file generator. The code weaver component reads this file to integrate selected feature source code with kernel source code.

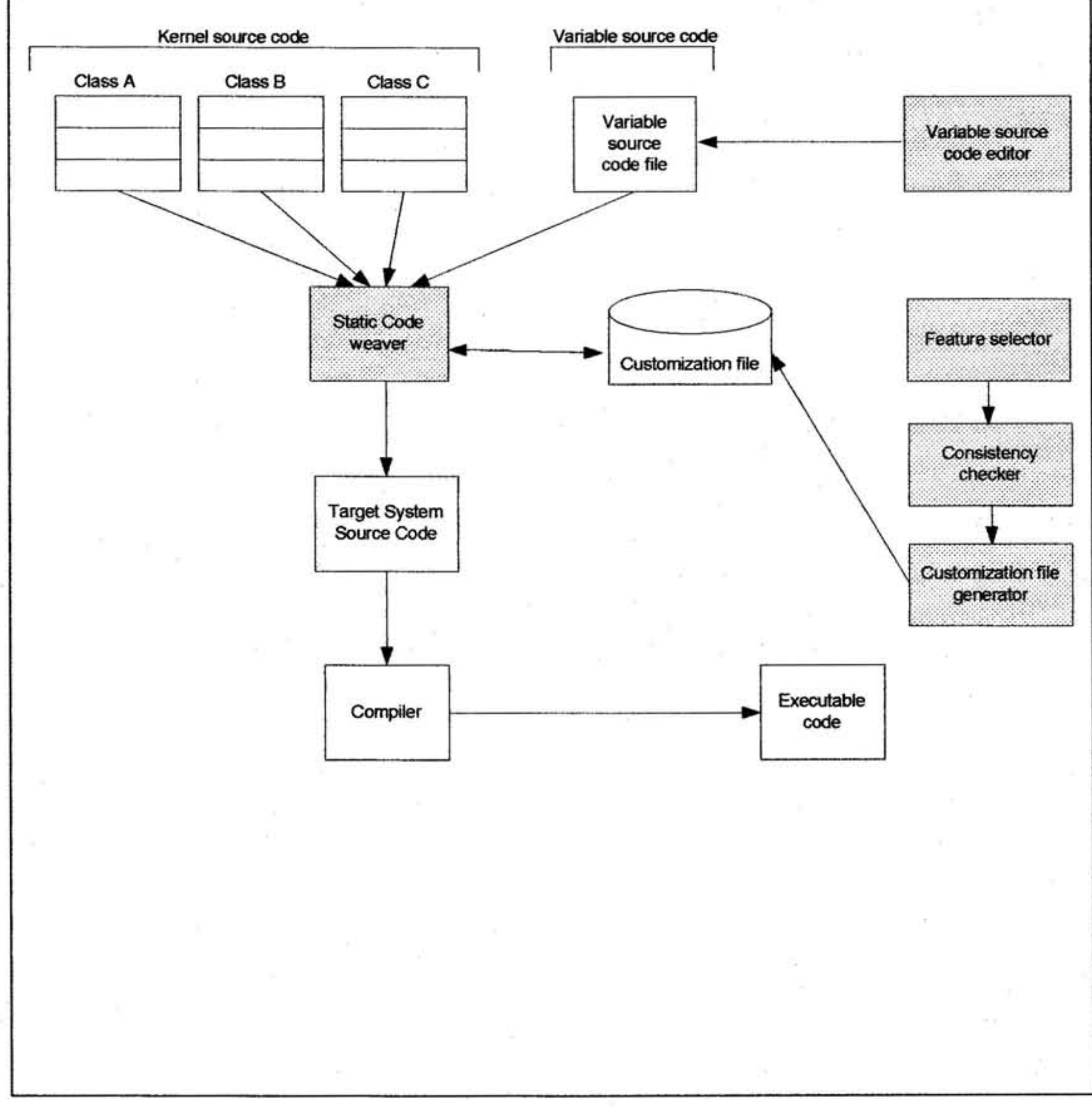
Step 3: Code weaving
This process combines kernel source code with optional and alternative source code from the created variable source code file and the customization file. This step is based on a source code integration engine, which reads the variable source code file code and inserts only selected source code that is related to selected features into the kernel source code at the specified insertion locations. This means, if an optional feature is selected, its related source code in the variable source code file will be inserted in the target system, and if one or the other alternative feature is selected, only related source code of the selected alternative feature is inserted in the target system at the location of the insertion point. Feature grouping and insertion points are the key for separation of concerns and source code integration.

(SCAC Pattern – Continue)

Dynamics

The following scenario depicts the dynamic behavior of code weaving step:

- Run the code weaver component.
- Read selected optional and alternative source code from the variable source code file and integrate it into kernel classes at the specified insertion point locations. The generated customization file is used for making decisions on which feature source code to insert.
- Compile integrated source code to generate an executable target system with only the required target system source code.



(SCAC Pattern – Continue)

The following diagram shows the complete processes of separation of concerns, feature selection, and code weaving:

Create Kernel source code in classes

Create client application classes

Kernel source code

Add Insertion points where source code from the variable source code file will be inserted based on feature selection

Add insertion points in classes

Create a variable source code file that contains source code related to alternative and optional features

Create variable source code file

Variable source code file

Select target system features and run consistency checks

Select features

Read selected features and integrate related source code from the feature file into kernel classes

Weave code

Kernel source code

Compile the integrated source code to generate an executable target system

Compile

Integrated source code

Run Executable target application

Executable code

(SCAC Pattern – Continue)

Step 4: Target application interaction
Once the interactive application is integrated and compiled, it will have the following components structure:

- User interface component
- Web service component

User interface component is responsible for accepting input from users and allowing invocation of possible service requests. It involves the sequencing of web services invocation and handling of message communication based on the customized workflow. It is also responsible for displaying results to users received from the web service component.

Web Service component is a collection of functional methods that are packaged as a single unit and published in the Internet for use by other software programs, in this case the user interface component.

Class Web service	**Collaboration**
Responsibility - Process a service request based on provided input - Returns results of processed request	- User interface

Class User interface	**Collaboration**
Responsibility - Accepts user input and service request - Invoke and pass parameters to appropriate web service(s) - Receives results from web service(s) - Display information to the user	- Web service

(SCAC Pattern – Continue)

Dynamics

The following scenario shows how service requests are processed using SCAC:

- User invokes a user interface
- User requests a service by entering input data and clicking a button
- User interface passes the service request and input data to a web service method(s).
- Web service processes request and returns results to the user interface. A web service may also request service from other web services.
- User interface displays results received from web service.

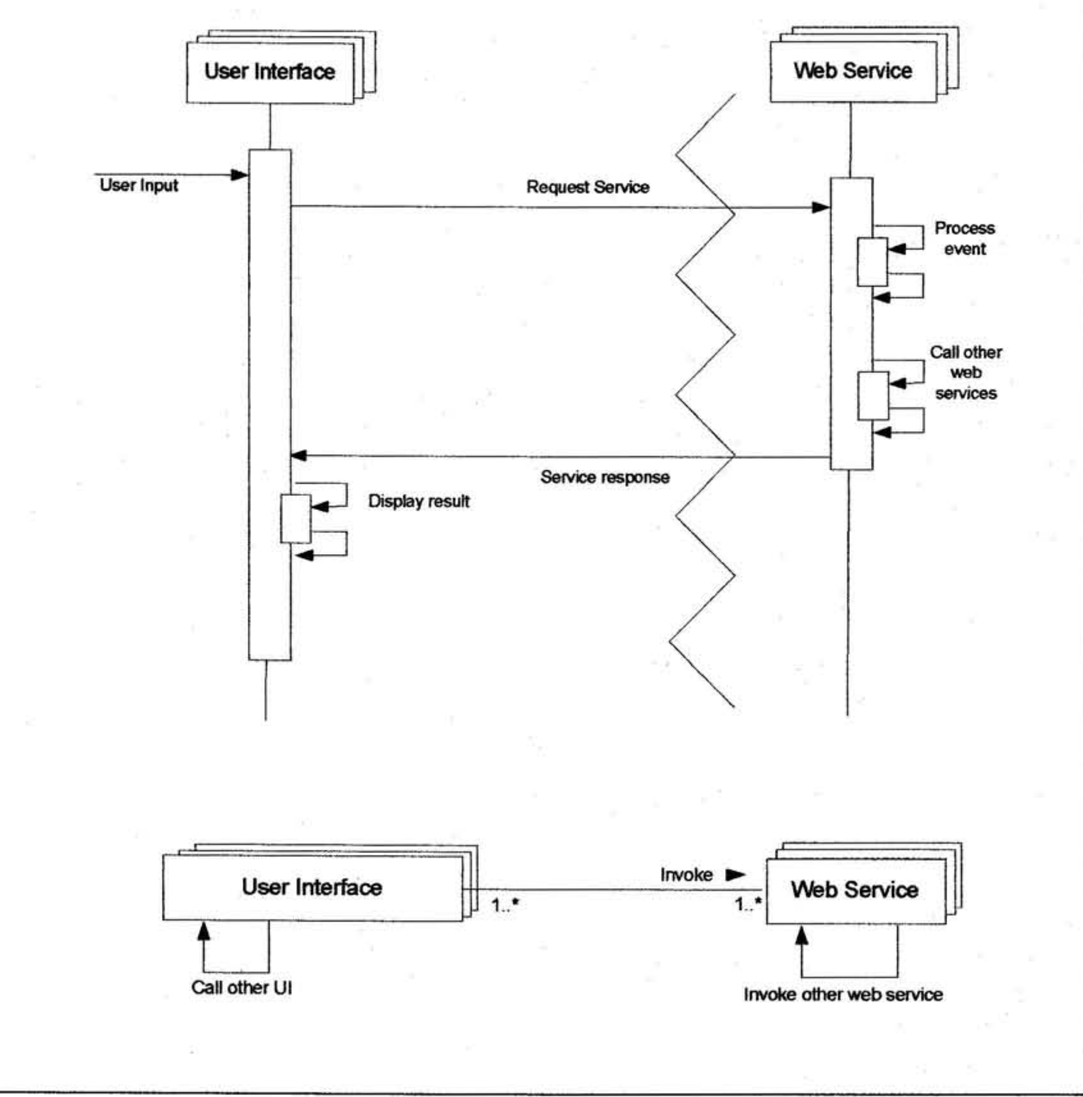


Figure 5-16 Static Client Application Customization (SCAC) Pattern

5.5.1 Development of SCAC pattern

This section describes the development of the SCAC pattern. The "MainReservation" UI from the hotel product line is used for this illustration. The ideas presented in this example can be applied to all other user interface objects. The example will illustrate the approach of separating optional and alternative source code from kernel source code into a variable source code file. It will also explain the integration of variable source code file code with kernel source code to produce a customized single target system. The separation and integration of source code will be based on the SCAC pattern, which includes the following activities: variable source code file creation, feature selection, consistency checking, and source code integration. Integrated source code is compiled to create an executable target system. The example follows the processes described in the SCAC pattern.

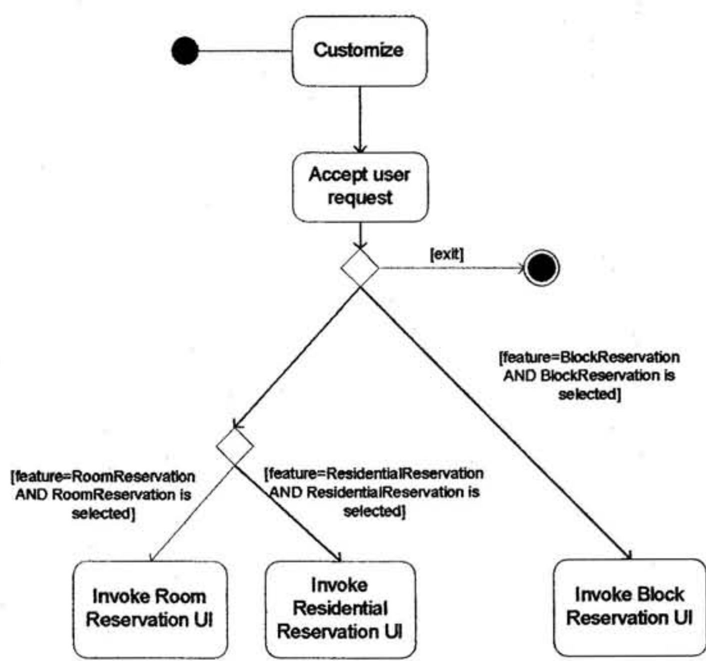


Figure 5-17 Activity Diagram - Main Reservation UI

Figure 5-17 shows the same customizable activity diagram used to illustrate the implementation of method 1 (DCAC) and method 2 (DCAC-SC). However, this example focuses on generating static client applications from a SPL system by extracting only required source code for running a target system. The customizable activity diagram represents the "MainReservation" user interface. It shows "ResidentialReservation" UI and "RoomReservation" UI as mutually exclusive alternatives where only one of them can be invoked by clicking the single reservation button of "MainReservation" user interface (Figure 5-18). "BlockReservation" UI, on the other hand, belongs to an optional feature. It will be either enabled or disabled based on whether BlockReservation feature is selected by the user. These decisions are made at source code integration time, unlike the DCAC and DCAC-SC patterns where decisions are made during application run time. In SCAC, only selected feature source code is extracted from the variable source code file and integrated with the kernel source code using the code weaver engine and the customization file.

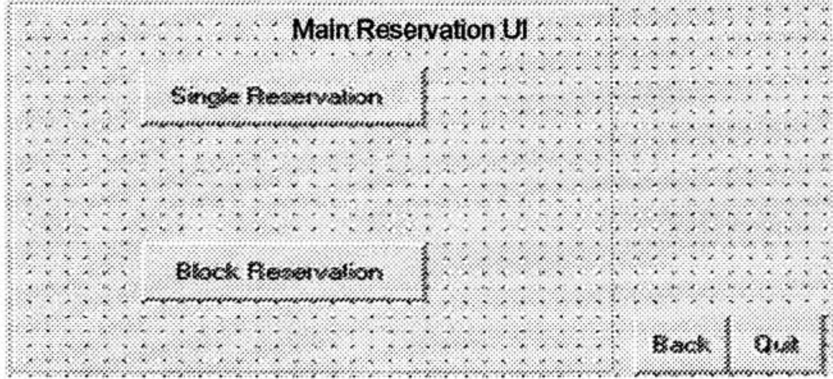


Figure 5-18 MainReservation graphical user interface

Figure 5-18 is the graphical user interface for "MainReservation" UI class. It shows two event buttons: Single reservation and Block Reservation. If single reservation button is clicked, either "RoomReservation" UI or "ResidentialReservation" UI will be invoked. Also BlockReservation button will either be visible or invisible based on feature selection and the SPL customization process.

Figure 5-19 shows both the kernel source code of the "MainReservation" UI and optional and alternative source code in the variable source code file. Insertion points are the key for integrating source code together. If an optional feature is selected, its related source code in the variable source code file will be inserted in the target system at the location of the insertion point. For example, the insertion point *$START BlockResButton* refers to the optional feature "BlockReservation" in the variable source code file. If this feature is selected, the related source code will be inserted in the kernel "MainReservation" UI class in the place of the insertion point: *$START BlockResButton*.

Also, if one or other alternative feature is selected, only source code related to the selected alternative feature is inserted in the target system at the location of the insertion point. For example, the insertion point *$START RoomResidentialUI* refers to the alternative features RoomReservation and ResidentialReservation in the variable source code file. Only one of the alternative source code blocks will be inserted. The code weaver engine will read the feature selection from the customization file and make the decision as to which source code block to insert.

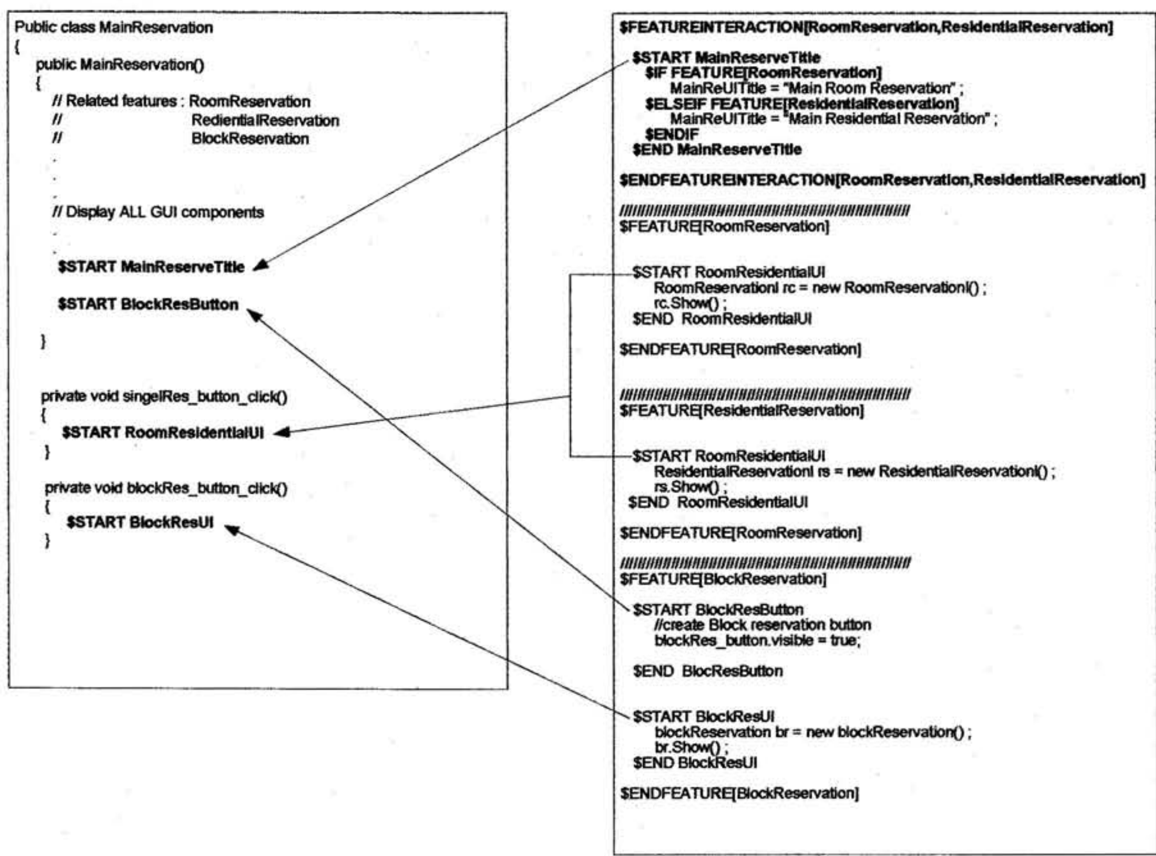


Figure 5-19 Implementation - Main Reservation UI

Feature dependency verification is handled during feature selection using the consistency checker component. The consistency checker verifies feature selection and notifies the user whether his selection is valid or not by consulting the feature model and applying consistency checking rules, described in Chapter 6. The generated customization file will then contain all selected and verified features. The code weaver engine reads the customization file and makes decisions on which source code blocks to insert. The command $FEATURE[<<feature name>>] is read by the code weaver engine and crosschecked with the customization file to verify whether the feature condition is set to

true or false. If it is set to true, the related source code block in the variable source code file is integrated with the kernel source code. Otherwise the source code block is ignored.

The $START RoomResidentialUI insertion point is used in both alternative feature groupings $FEATURE[RoomReservation] and $FEATURE[ResidentialReservation]. In the customization file, only one of the alternative features will be set to true in the feature selection process (mutually exclusive features), and only the variable source code block that is related to the selected feature is integrated with the kernel source code.

The command $FEATUREINTERACTION[<<feature 1, feature 2, ..>>] groups related variable source code that require a decision on which source code to include in the integration process. It is used when several features are affected by the feature selection decision. $FEATUREINTERACTION[RoomReservation, ResidentialReservation] is used to group the mutually exclusive alternative features RoomReservation and ResidentialReservation in one decision block. The decision commands $IF FEATURE[RoomReservation] and $ELSEIF FEATURE [ResidentialReservation] are used to specify which source code block to integrate based on feature selection in the customization file. The decision commands provide flexibility to developers during source code construction time to specify different actions that are required when two or more features are selected within the feature interaction block. For example:

```
$IF FEATURE[feature 1, feature 2]
        // insert code A
$ELSEF FEATURE [feature 1]
        // insert code B
$ELSEF FEATURE [feature 2]
        // insert code C
```

The above source code means that if feature 1 and feature 2 are selected then insert source code A. If only feature 1 or feature 2 is selected then insert either source code B or C blocks.

The variable source code file in Figure 5-19 can also be rewritten to include alternative features RoomReservation and ResidentialReservation in one feature interaction grouping rather than two different feature groupings as shown in Figure 5-19. The sample source code in figure 5-20 shows the structure of the grouping:

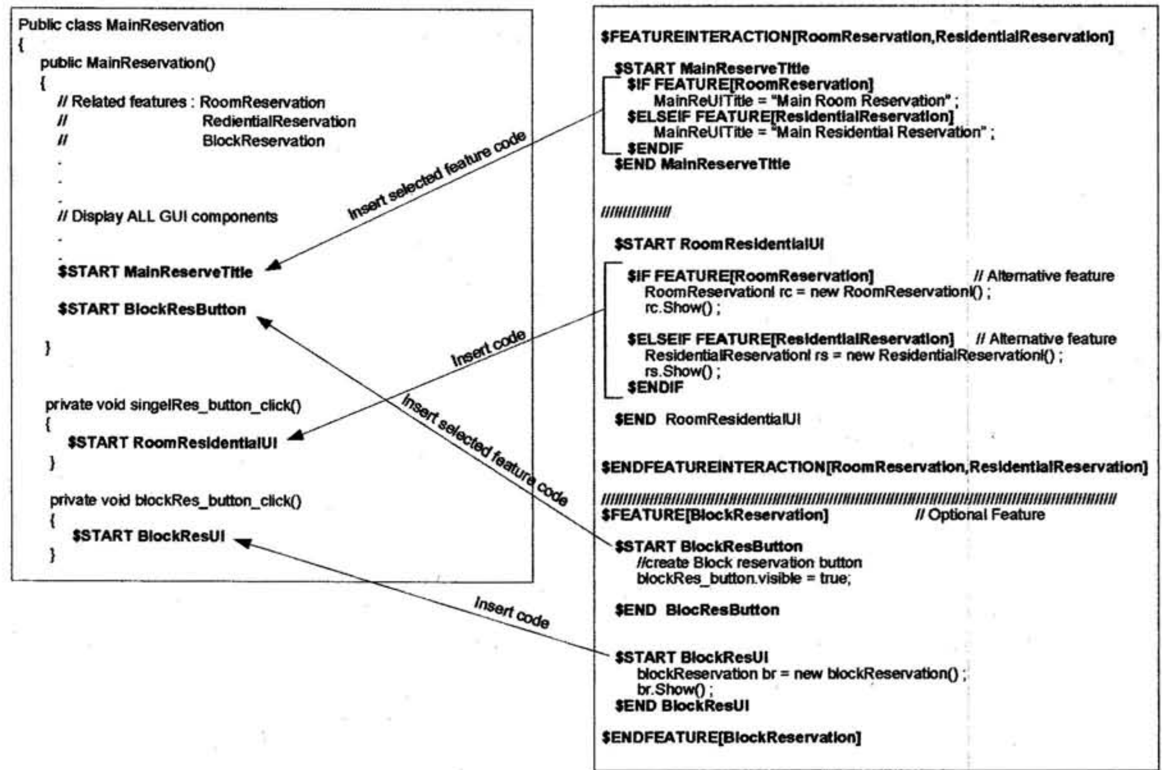


Figure 5-20 Implementation - Main Reservation UI

Even though in this case there is no different action required when two or more features are selected because the two features are mutually exclusive, Figure 5-20 demonstrates the possibility of combining related variable source code blocks into one feature interaction block. This is an implementation decision that is left to the developer. However, if different actions are required when two or more features are selected, feature interaction grouping is mandatory to enable the code weaver engine to make the decision on which source code block to integrate.

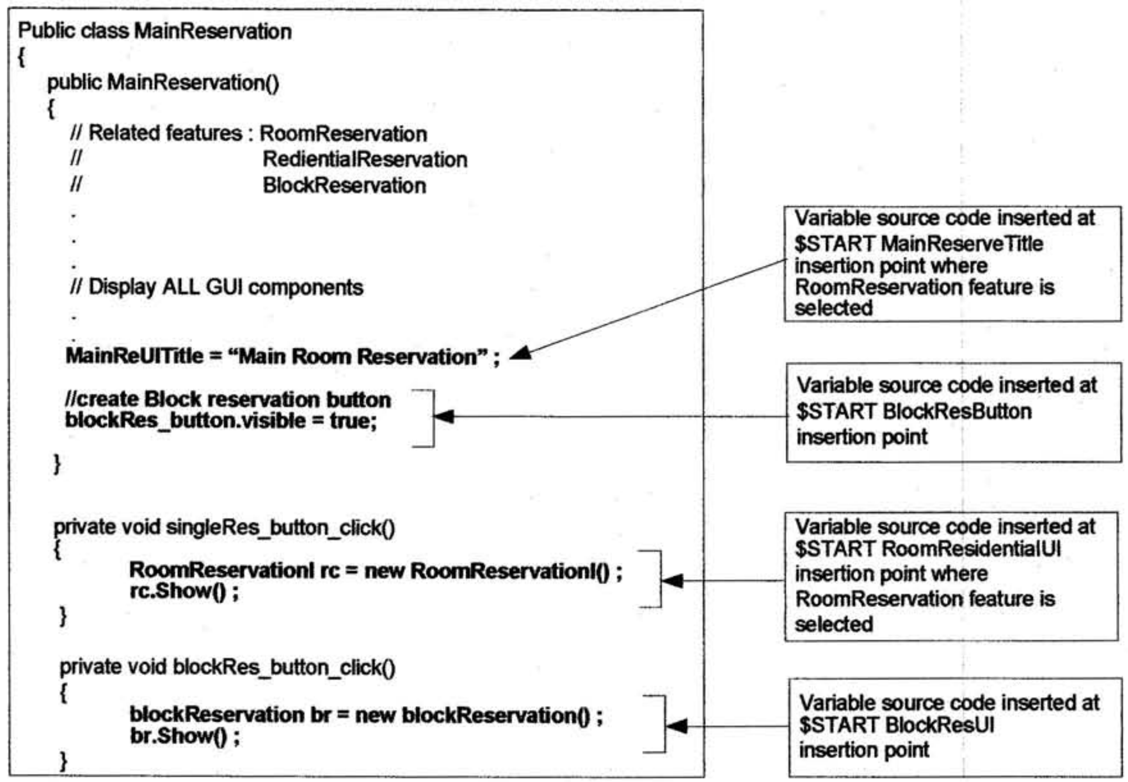


Figure 5-21 Implementation - Main Reservation UI with RoomReservation feature

Whether the variable source code file structure of Figure 5-19 or Figure 5-20 is used, the result of the integration process will be the same. Figure 5-21 shows the "MainReservation" UI class after the integration process. In this class, the BlockReservation feature and the RoomReservation feature are inserted in the kernel source code. Inserted blocks are:

- Insertion point $START MainReserveTitle in the kernel source code is replaced with the following source code from the variable source code file:

  MainReUITitle = "Main Room Reservation" ;

- Insertion point $START BlockResButton in the kernel source code is replaced with the following source code from the variable source code file:

// Create block reservation button
blockRes_button.visible = true;

- Insertion point $START RoomResidentialUI in the kernel source code is replaced with the following source code from the variable source code file:

RoomReservationl rc = new RoomReservationl() ;
rc.Show() ;

- Insertion point $START BlockResUI in the kernel source code is replaced with the following source code from the variable source code file:

blockReservation br = new blockReservation() ;
br.Show() ;

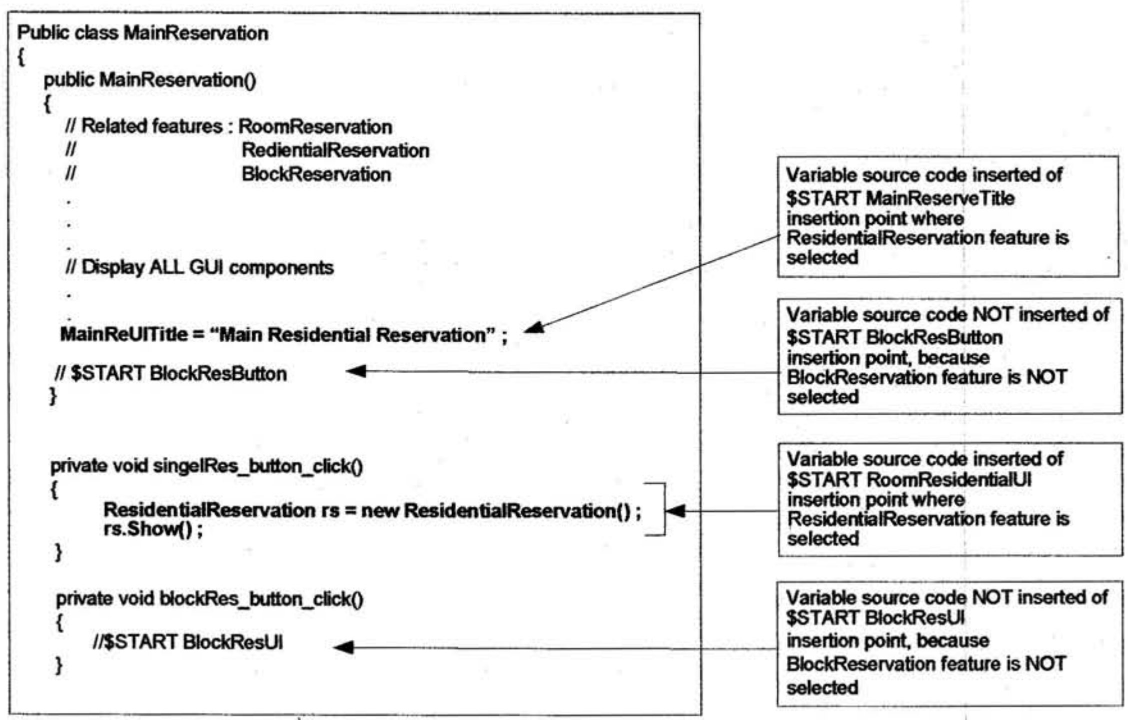


Figure 5-22 Implementation - Main Reservation UI with ResidentialReservation feature

Figure 5-22 shows the "MainReservation" UI class after the integration process. In this class, the ResidentialReservation feature is selected but not the BlockReservation feature. Therefore, only the source code related to ResidentialReservation feature is inserted in the kernel source code. Inserted blocks are:

- Insertion point $START MainReserveTitle in the kernel source code is replaced with the following source code from the variable source code file:

  MainReUITitle = "Main Residetial Reservation" ;

- Insertion point $START RoomResidentialUI in the kernel source code is replaced with the following source code from the variable source code file:

  ResidentialReservationl rs = new ResidentialReservationl() ;
  rs.Show() ;

The implementation of the insertion points in the variable source code file of this method differs from the DCAC-SC. In the DCAC-SC pattern, insertion points have no decision conditions to tell the code weaver engine what part of the source code to include or ignore. In the SCAC method, the IF FEATURE and ELSEIF FEATURE decision statements are used to extract only selected variable source code and perform the integration with kernel source code. The SCAC approach is suitable for SPL applications that require distribution of only needed target application source code.

5.5.2 Advantages of SCAC approach:

The advantages of the SCAC are similar to DCAC and DCAC-SC methods regarding the use of service-oriented architecture in developing software product lines. However this method has a different advantage at the SPL customization phase. In the SCAC method, only selected variable source code is extracted from the variable source code file and integrated with the kernel source code, which means elimination of source code overhead. Static workflows are produced to eliminate source code overhead for decisions made during run time as to what optional or alternative source code to execute, which are required in the DCAC and DCAC-SC approaches. All integrated source code in the SCAC approach will be used in the target system.

5.5.3 Disadvantages of SCAC approach:

- Source code extraction is required for each target system.
- Target system has to be compiled every time a target system is customized.

5.6 Comparison of customization methods

Table 5-1 is a comparison of the three customization methods. It shows all the characteristics of each method compared to the other two methods.

	Method 1 (DCAC)	Method 2 (DCAC-SC)	Method 3 (SCAC)
Customization	At run time	At run time	At code weaving time
Insertion points	No	Yes	Yes
Separation of common and variable source code	No	Yes	Yes
Application code	All optional and alternative source code is included	All optional and alternative source code is included	Source code specific to target application
Customization of application workflows	Dynamic customization	Dynamic customization	Static customization
Common & variable source Code integration	No integration	Once	Every time a target application is customized
Compilation	Once	Once	Every time a target application is customized
Feature interaction in application code	Variable source code and feature decisions are intertwined	Variable source code and feature decisions are intertwined	No feature decisions in target code

Table 5-1 Comparison of the Three Patterns

5.7 Usage of Development Approaches

Based on the advantages and disadvantages of each approach, a development approach can be selected. The DCAC development approach enables developers to build a dynamically customizable application that can be customized at run time. Once the SPL application is developed and compiled, modification to source code of derived target applications is not required. Application engineers can select desired features for a target application and generate a customization file to be used at run time.

The DCAC-SC approach is developed to address the issue of separation of concerns related to the dynamic customization of client applications (DCAC). Developers may use this approach for better maintenance of SPL applications by separating variable source code from kernel source code into a variable source code file. However, this approach requires more work for developers to do the separation and integration of variable source code and kernel source code.

The SCAC approach is suitable for SPL applications that require extracting only needed source code to run target applications. This approach eliminates code overhead, which is required in the first two approaches for making decisions as to what optional or alternative source code blocks to execute during run time. In this approach, all integrated source code of selected features will be used in the target system. However, the SCAC approach requires source code integration and compilation for every customized target application.

5.8 Summary

This chapter has described three development and customization methods to configure applications from a software product line that is based on service-oriented architecture: dynamic customization of client application, dynamic customization of client application with separation of concerns, and static customization of client application with separation of concerns. A design pattern was used to describe each development and customization method. Activity diagrams, screenshots, collaboration diagrams and source code samples illustrate the development of each approach in the context of alternative and optional feature selection for applications derived from a software product line.

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

6. SOFTWARE PRODUCT LINE ENVIRONMENT PROTOTYPE

6.1 Introduction

This chapter describes the Software Product Line Environment Prototype (SPLET) as a proof of concept for this research. It is based on the PLUS environment of the Evolutionary Software Product Line Engineering Process in Figure 6-1 [Gomaa00,Gomaa04]. SPLET is a SPL independent prototype that is designed to cover the SPL environment. It covers the software product line engineering phase and the application engineering phase (SPL customization).

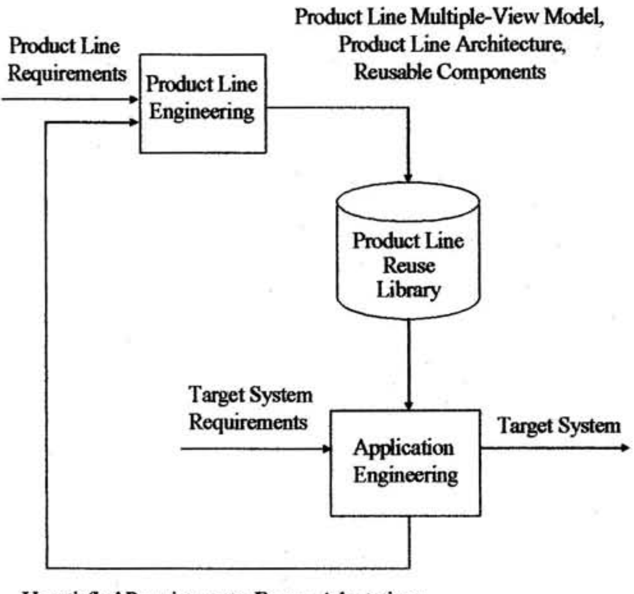


Figure 6-1 Evolutionary Software Product Line Engineering Process

During the software product line engineering phase, SPLET enables SPL engineers to store links to all design models, architectures, and application components in the reuse library for the purpose of navigating between the multiple-view models and testing web service components. In this phase a facility is provided to enable the creation of a SPL Model that organizes all SPL engineering components by their related features. The SPL Model is used as the main driver for customizing the SPL application in the next phase.

The application engineering phase is addressed in SPLET through the provided facilities that enable application engineers to select desired features, run consistency checking rules, and customize target applications using one of the three implementation approaches, described in Chapter 5.

6.2 Software Product Line Environment Prototype (SPLET)

The software product line environment prototype is a domain independent prototype that covers the entire SPL life cycle. It is designed to support most popular languages such as C, C++, C#, JAVA, and J++.

SPLET prototype is based on organizing a SPL into features that are categorized as kernel, optional, and alternative. Features are the main driver for organizing SPL components and customizing target applications. Each feature in the SPL Model stores links to all related designs, architecture, and implementation components. The SPLET prototype helps visualize the overall SPL by providing a flexible navigation facility

through the SPL Model, and provides the needed facilities to customize target applications.

SPLET prototype includes the following components:

- SPL feature editor:
    - Allows SPL engineers to create a feature dependency tree and defines feature relations.
    - Allows SPL engineers to create parameterized variables for each parameterized feature.
    - Allows SPL engineers to define mappings between features and related web service components.
    - Allows SPL engineers to define mappings between features and related artifacts, such as specifications, designs, and test procedures.
- Web service editor:
    - Allows SPL engineers to enter web service components and link them to their location on the Internet. The entered web service list is used by the SPL engineers to map web services to features using the feature editor component.
- Feature selector:
    - Allows application engineers to select desired features
    - Allows application engineers to enter values for parameterized variables

- Consistency checker: This component is part of the feature editor. It serves as a checker for ensuring that selected features are consistent with each other. When a feature is selected, the consistency checker is invoked to verify selection.
- Customization file generator: This component is responsible for automatically generating a customization file that is required for the dynamic customization of client applications at system run time. It is based on the feature selector component. It sets the selection status of each feature to true/false and stores the values of parameterized variables.
- Variable source code editor: Creates a variable source code file that stores related optional and alternative source code for each feature to be used in the source code integration process.
- Code tracker: Locates insertion source code in the variable source code file and kernel source code by features.
- Code weaver: This component is used for the source code integration process. It is responsible for integrating kernel source code with optional and alternative source code using the automatically generated variable source code file and feature selection.
- File extractor: This component is used to retrieve specifications, designs, source code, and test procedures for the selected features.

Figure 6-2 summarizes the proof-of-concept prototype SPLET.

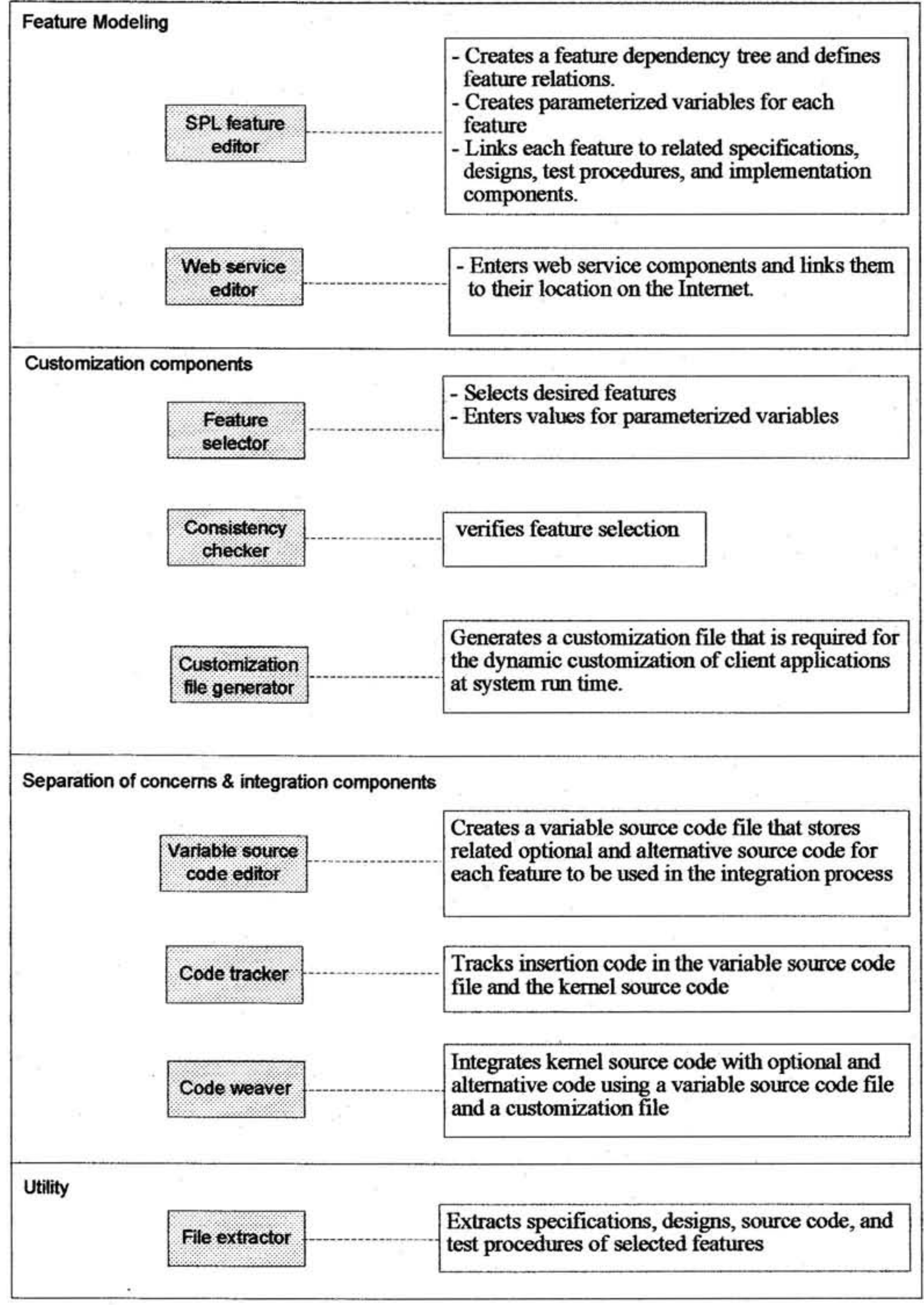


Figure 6-2 SPLET components

Figure 6-3 shows a detailed description of SPLET components. It consists of four subsystems: feature modeling, customization, separation of concerns, and supporting utility components.

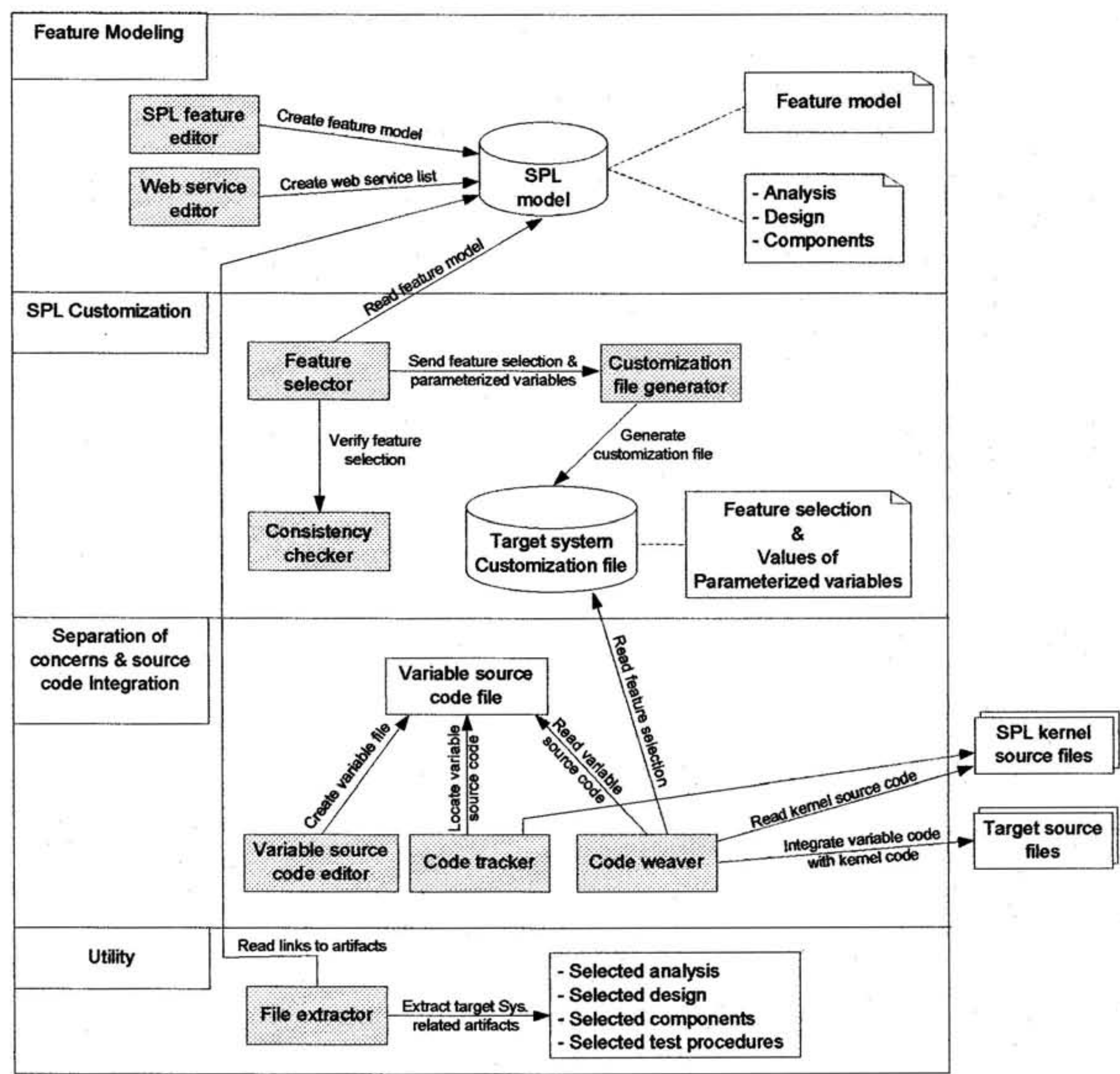


Figure 6-3 Detailed description of SPLET

Figure 6-4 shows the main screen of SPLET with its division of subsystems shown in Figure 6-3. The next section describes in detail each subsystem and its related components.

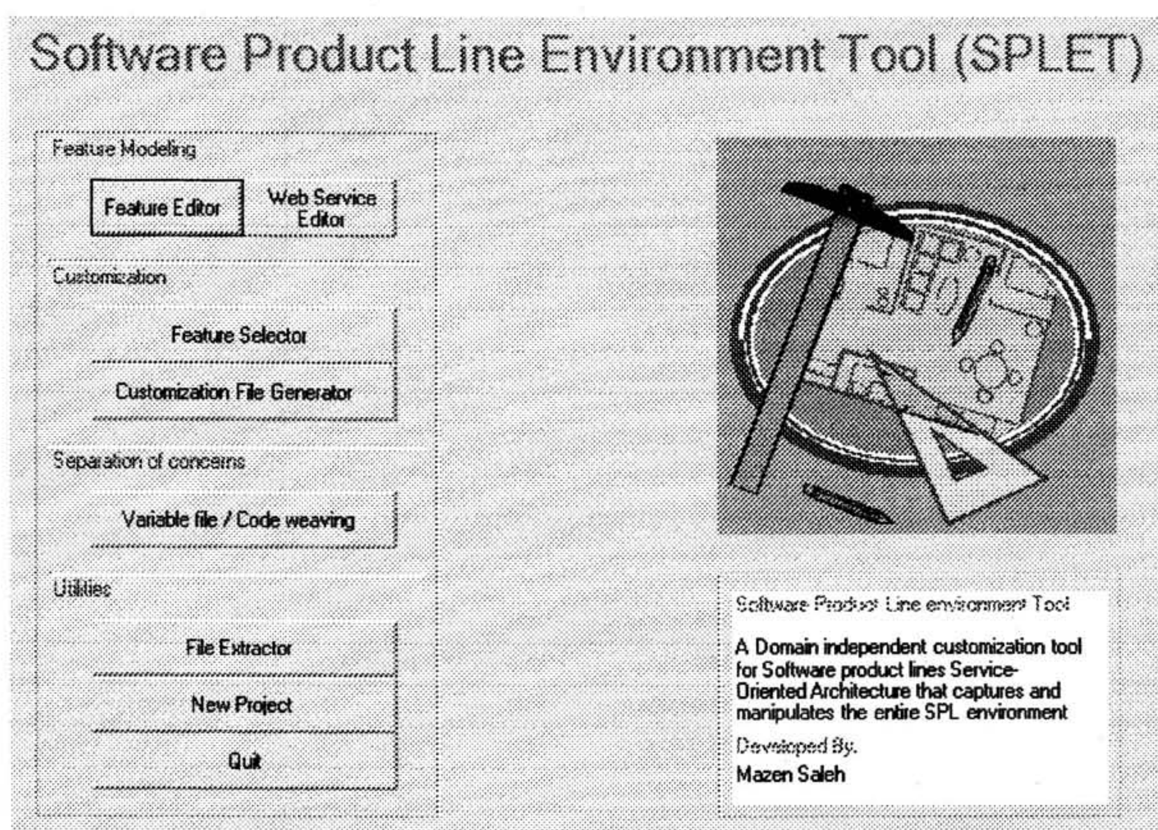


Figure 6-4 SPLET - Main Screen

6.2.1 Feature Modeling Subsystem:

This subsystem consists of two components: SPL Feature Editor and Web Service Editor. The two components interact with the SPL Model database for creating the feature model and all product line artifacts. Product line artifacts consist of specifications, designs, web

service components, source code, and test procedures. This subsystem is the basis for all other subsystems. Figure 6-5 shows the subsystem and the interaction between the components and the SPL model database. The subsystem consists of two components: SPL Feature Editor and Web Service Editor. The SPL feature editor component is used to create the feature model as a hierarchical feature tree. It also associates the SPL artifacts to their related features. The web service editor component is used to enter all needed web services for the SPL application. The SPL model database is the reuse library for the SPL environment.

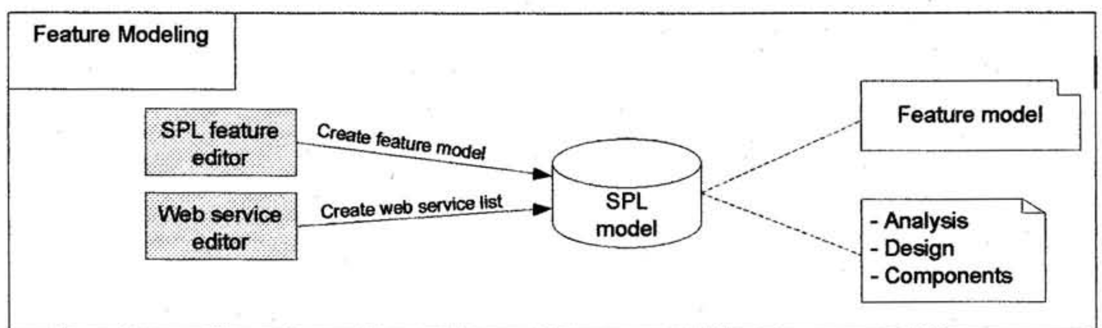


Figure 6-5 Feature Modeling Subsystem

Figure 6-6 shows the entity class diagram for the SPL model. It consists of the following entity classes:

MainFeatureSelection: Stores all feature names, their description, features type (kernel, optional, alternative), and feature grouping names of related alternatives.

Variable: Stores parameterized variables for each parameterized feature.

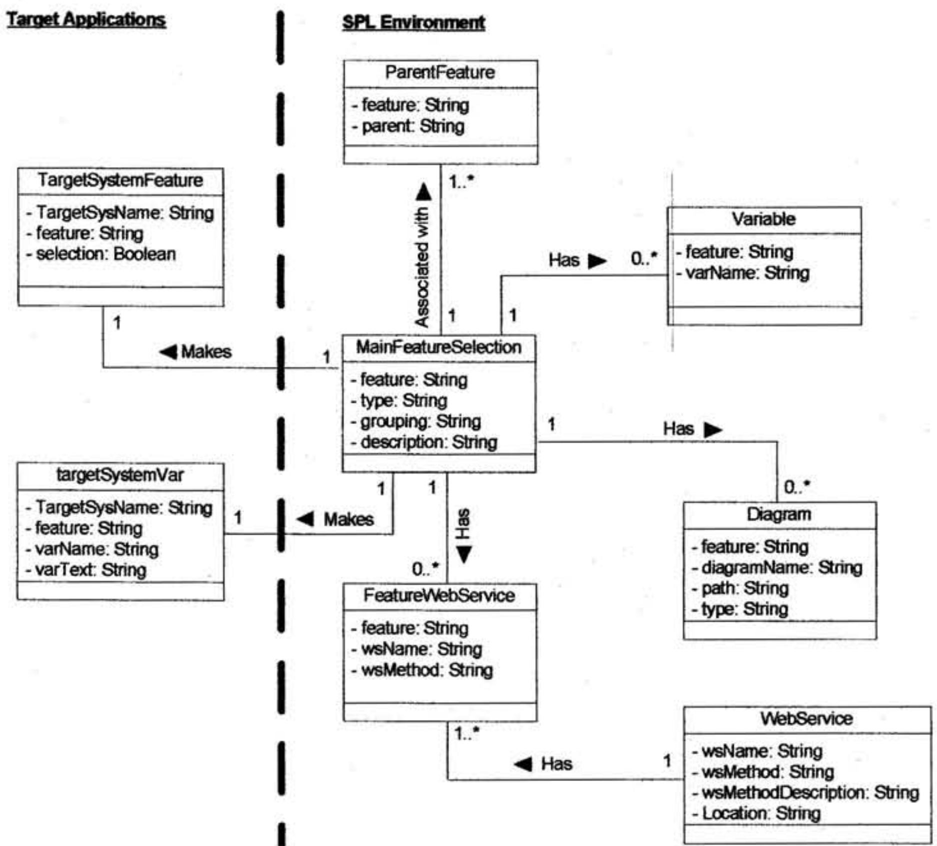


Figure 6-6 Entity Class Diagram

ParentFeature: Stores the parents of each feature to create the feature tree required in the customization and consistency checking processes.

Diagram: Stores feature name, diagram name, and diagram path (storage location) of all analysis, design, source code files, and testing procedures for the purpose of navigating through the multiple-view model. Diagrams are categorized by type (analysis, design, source code, and tests) in the type field.

FeatureWebService: Stores feature name, web service name, and related web service methods to enable the invocation of web services in the SPLET prototype, for the purpose of testing their behavior and their required input and output.

WebService: Stores all web service names, related web service methods, description of each web service method, and their URL location in the Internet.

TargetSystemsFeature: Stores features and their selection status for each target application.

TargetSystemVar: Stores features and values of related parameterized variables for all target applications.

The following describes the two components of the SPL Feature Modeling subsystem: Feature Editor and Web Service Editor.

6.2.1.1 Feature Editor Component

This component is used to create the SPL model. It provides the following facilities:

- Allows SPL engineers to create a feature tree and defines feature relations.
- Allows SPL engineers to create parameterized variables for each feature
- Allows SPL engineers to define mappings between features and related specifications, designs, test procedures, and implementation components.

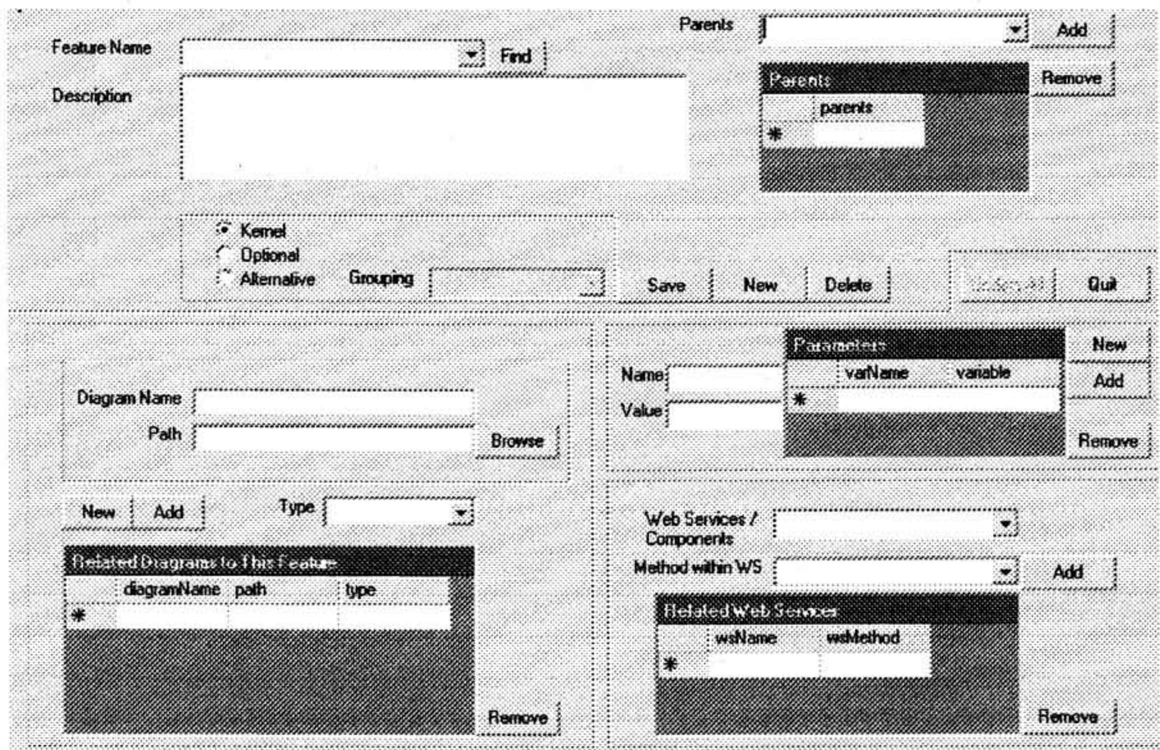


Figure 6-7 Feature Editor-main interface

The main user interface of the Feature Editor component in Figure 6-7 is decomposed into several screen snapshots to describe each part of the main user interface.

The Feature Editor user interface of Figure 6-8 is used to create features and associate features to their parents to produce a feature tree.

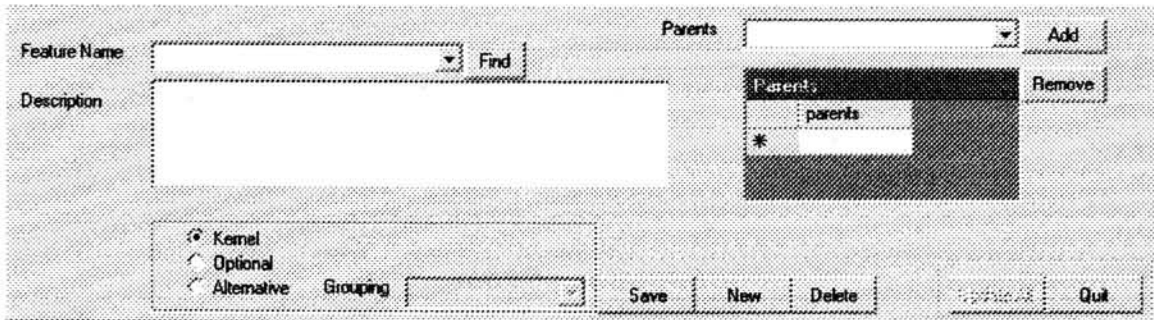


Figure 6-8 Feature Editor – feature creation

Figure 6-9 shows the sequence diagram for creating features in the Feature Editor component.

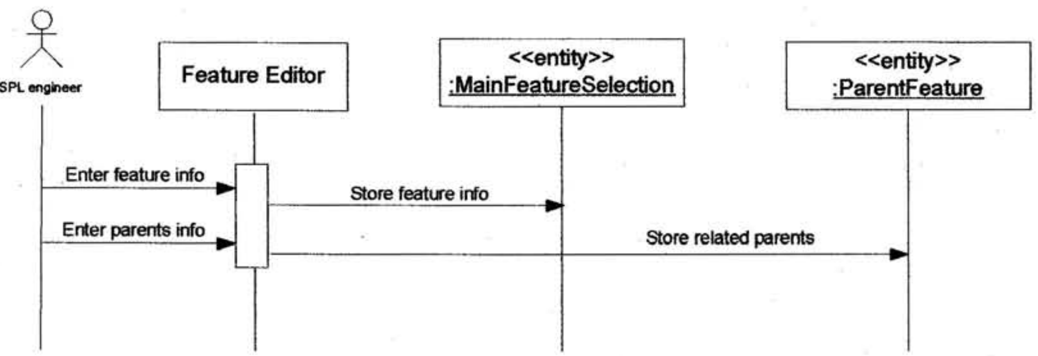


Figure 6-9 Feature Creation

The following explains the process:

- SPL engineer enters feature name, type (kernel, optional, alternative), description, and grouping name of alternative features.
- SPL engineer enters associated feature parents.
- The Feature Editor component stores entered information in the MainFeatureSelection and ParentFeature tables of the SPL model database.

Features are created from top of the tree to bottom. The top level feature is the main parent feature. Under the main feature, features under different levels of the hierarchy are created and associated with related parent(s) to form the feature tree. From the design model of Chapter 4, Figure 6-10 shows the feature tree used in the hotel system. In the top level, the main feature forms the root of the tree. The other features may have one or more parents associated with them. For example, "BlockCheckout" feature has "Checkout" and "Blockreservation" features as its parents.

For alternative features, the feature grouping "Reserve" is used to group the mutually exclusive alternative features "RoomReservation" and "ResidentialReservation". In the grouping field of Figure 6-8, the "Reserve" feature grouping is entered. The group name of the next related alternative feature created with the Feature Editor can be selected from the dropdown list of groupings.

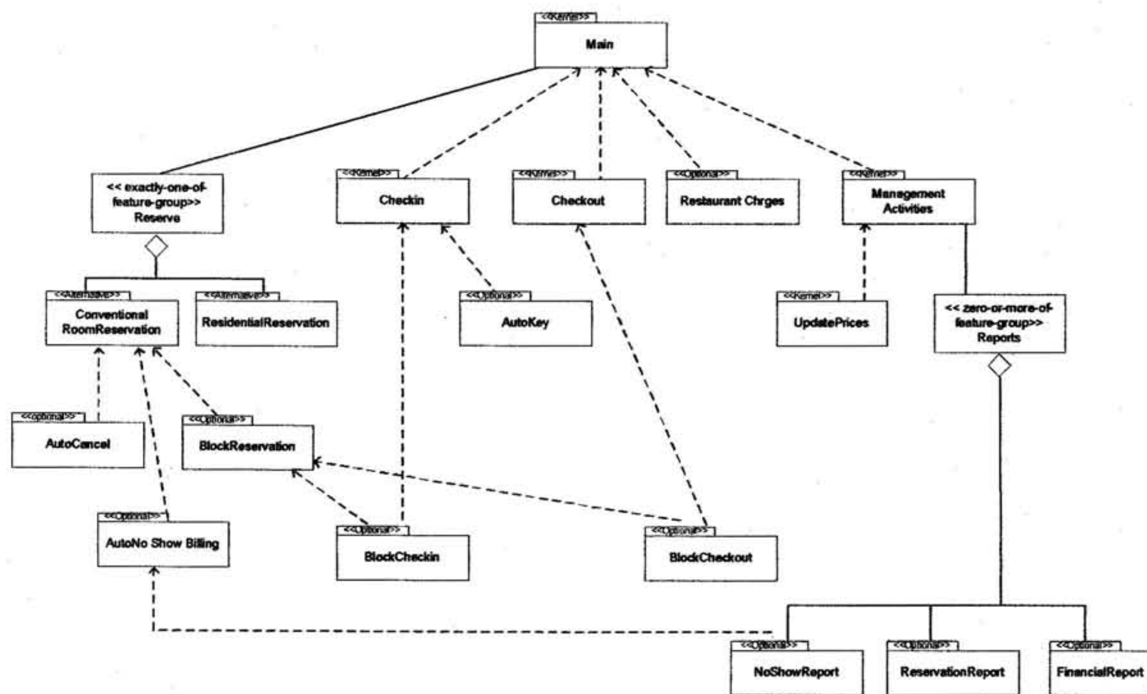


Figure 6-10 Feature dependency tree

The Feature Editor user interface of Figure 6-11 is used to define mappings between features and related specifications, designs, test procedures, and implementation components.

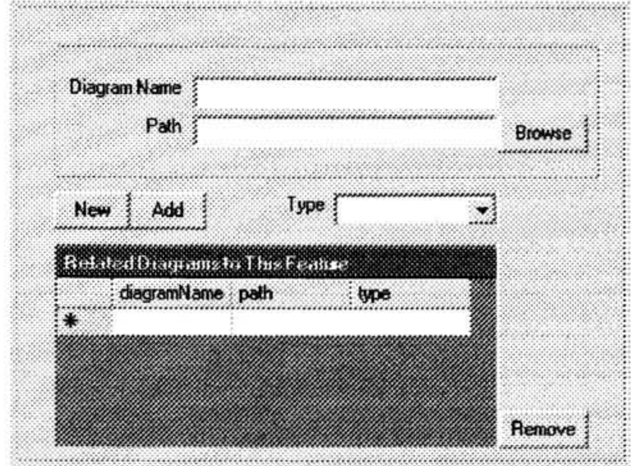


Figure 6-11 Feature Editor – related diagrams

Figure 6-12 shows the sequence diagram for associating SPL artifacts to features in the Feature Editor component.

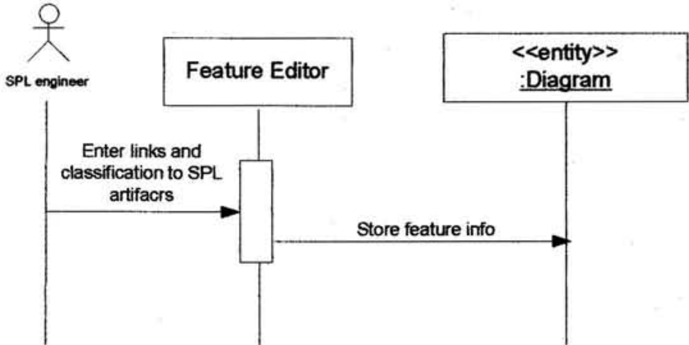


Figure 6-12 Storing related SPL artifacts

The following explains the process:

- SPL engineer enters links to associated SPL artifacts to each feature (specifications, designs, source code files, test procedures), and selects the proper classification of the artifact from the dropdown list of classification types.
- The Feature Editor component stores entered information in the Diagram table of the SPL model database.

The Feature Editor user interface of Figure 6-13 is used to create parameterized variables related to each feature. During customization, the values of parameterized variables are entered using the Feature Selector component under the customization subsystem. The customization file generator component extracts this information from the SPL model database.

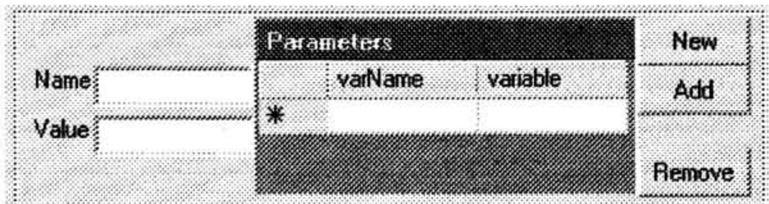


Figure 6-13 Feature Editor – parameterized variables

Figure 6-14 shows the sequence diagram for creating parameterized variables in the Feature Editor component.

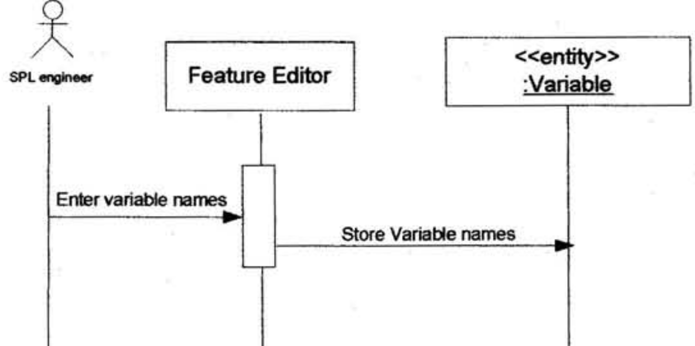


Figure 6-14 Creation of parameterized variable

The following explains the process:

- SPL engineer enters variable names to each parameterized feature.
- The Feature Editor component stores entered information in the variable table of the SPL model database.

The Feature Editor user interface of Figure 6-15 is used to relate web services to each feature. This information is used in the Feature Selector component under the customization subsystem to enable the invocation of web services for the purpose of testing their behavior and their required input and output.

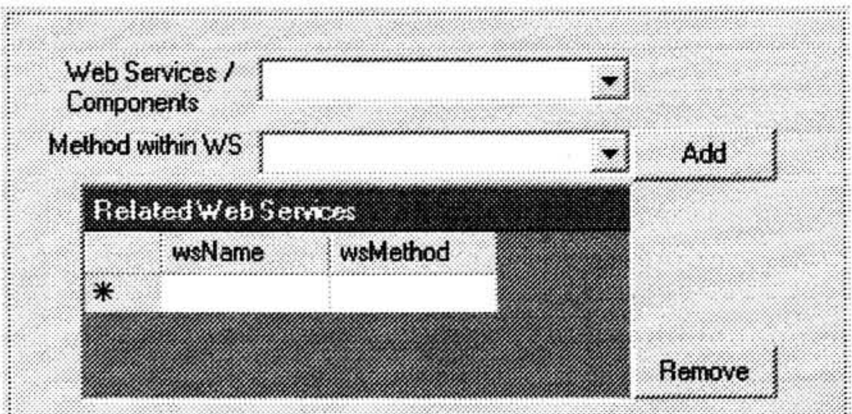


Figure 6-15 Feature Editor - web services

Figure 6-16 shows the sequence diagram for associating related web services to a feature using the Feature Editor component.

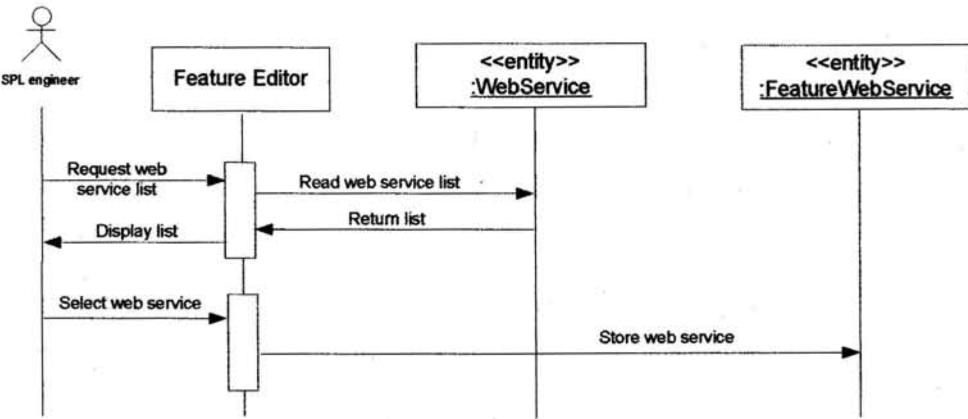


Figure 6-16 Adding web services

The following explains the process:

- SPL engineer requests a web service list from the Feature Editor component.
- The Feature Editor component reads the web service list from the WebService table.
- The web service list is returned to the Feature Editor component and displayed to the user.
- SPL engineer selects feature related web services from the list.
- Selected web services are stored in the FeatureWebService table.

6.2.1.2 Web Service Editor

The Web Service Editor is part of the Feature Modeling subsystem. Figure 6-17 shows the interface for this component. It is used to enter all needed web services related to the SPL application. The information is stored in the WebService table of the SPL model database. It is used in the Feature Editor component to select web services from this list. It is also used by the Feature Selector component to invoke web service methods.

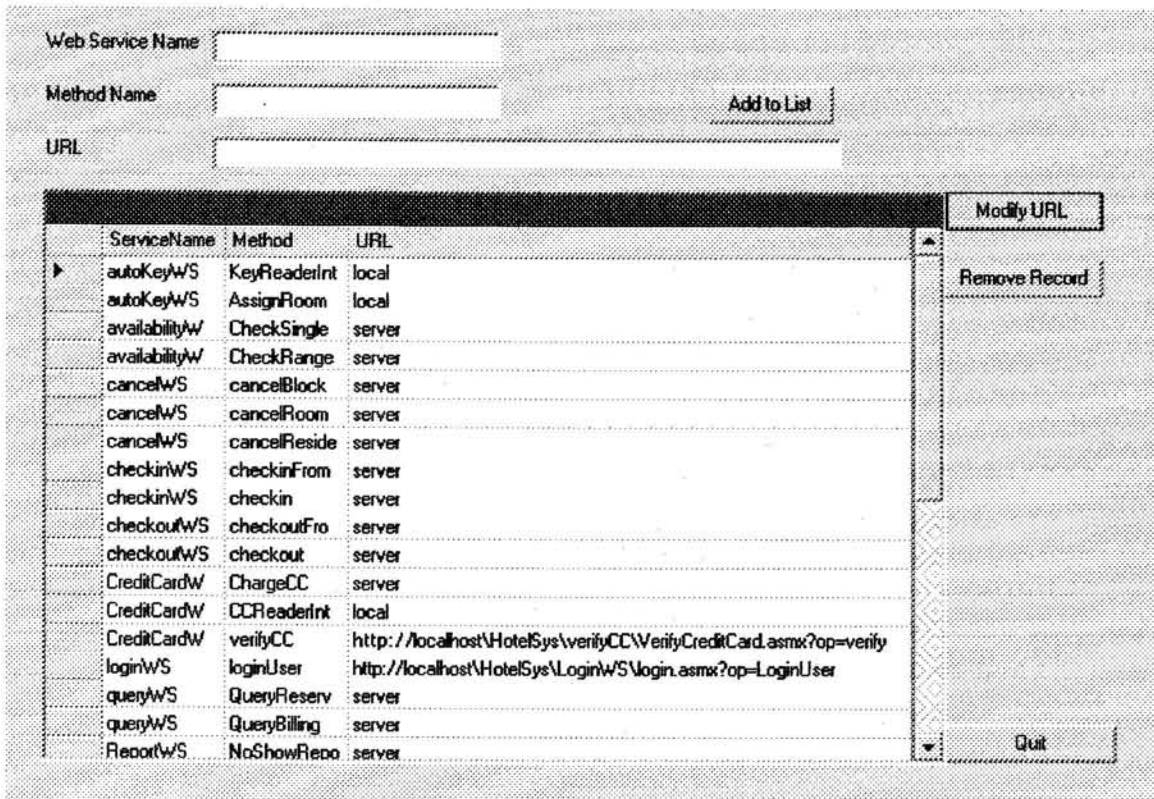


Figure 6-17 Web service editor

Figure 6-18 shows the sequence diagram for entering needed web services that are related to the SPL application.

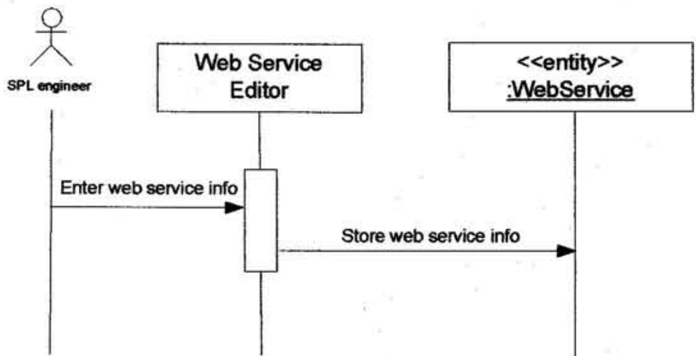


Figure 6-18 Adding web services

The following explains the process:

- SPL engineer enters web service information in the Web Service Editor component.
- The Web Service Editor component stores entered information in the WebService table of the SPL model database.

6.2.2 Customization subsystem:

This subsystem is used to customize target applications. It consists of three components: Feature Selector, Consistency Checker, and customization file generator. The Feature Selector and Consistency Checker components interact with the SPL model database for selecting features, entering values for parameterized variables, and verifying selected features. The customization file generator component generates the customization file used by the dynamic client application to enable customization at run time. Figure 6-19 shows the subsystem and the interaction between the components and the databases.

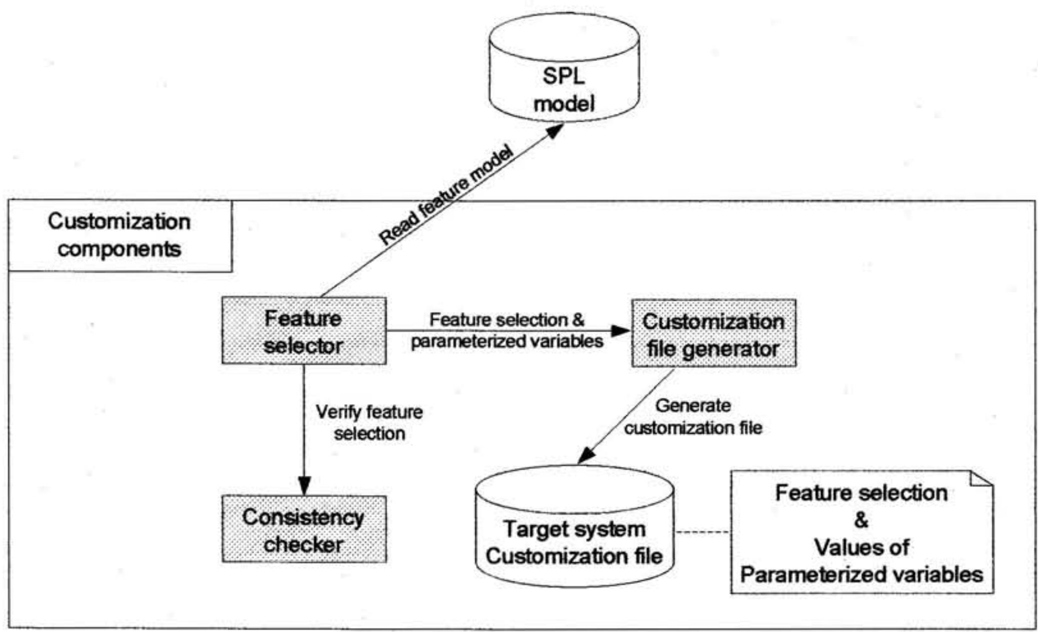


Figure 6-19 Customization Subsystem in SPLET

The following describes the three components of the customization subsystem: Feature Selector, Consistency Checker, and customization File Generator.

6.2.2.1 Feature Selector

This component is used to customize target applications and navigate through the feature tree. Figure 6-20 shows the main user interface of the Feature Selector component. This component provides the following facilities:

- Allows application engineers to select target application features.
- Allows application engineers to enter values of parameterized variables of selected features.
- Allows application engineers to navigate through the SPL environment to:

- View feature selection.
- View specifications, analysis, design, and source code files related to each feature.
- Invoke web service methods related to each feature for testing web services behavior and their input and output.

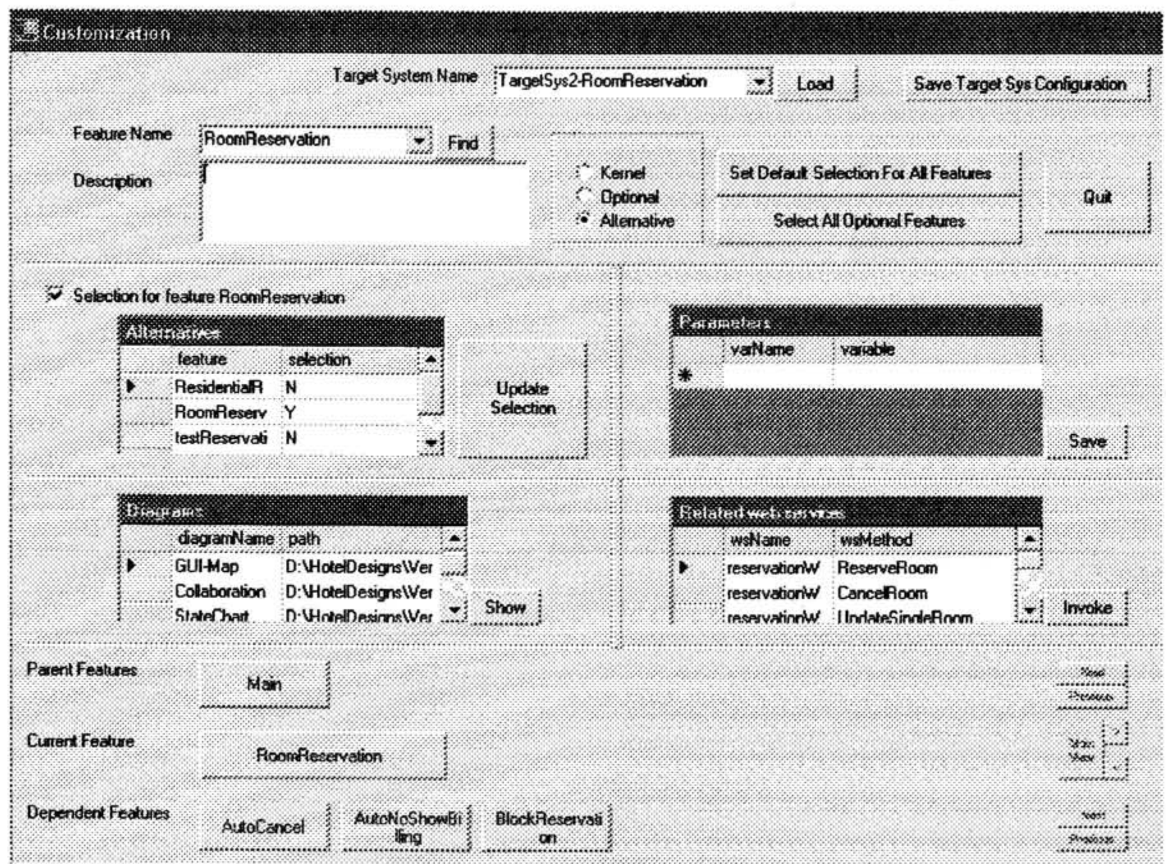


Figure 6-20 Feature Selector main interface

Figure 6-21 shows the sequence diagram for customizing target applications.

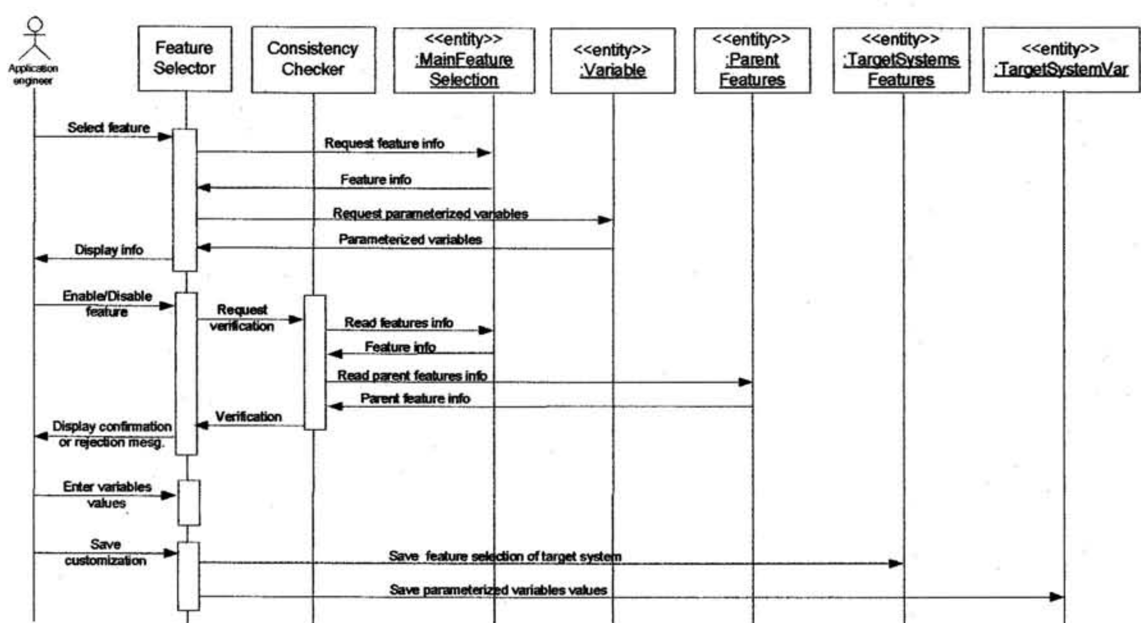


Figure 6-21 Feature Selector - Customization

The following explains the process:

- Application engineer selects a feature to be customized.
- Feature Selector component requests feature information from the MainFeatureSelection table of the SPL model database.
- Feature Selector component requests related parameterized variables from the Variable table of the SPL model database.
- Feature Selector component displays feature information and related parameterized variables.
- Application engineer requests from the Feature selector component to enable or disable the selected feature.

- Feature Selector component requests from the Consistency Checker component verification to enable or disable the selected feature.
- Consistency Checker component reads the MainFeatureSelection table and the ParentFeature table.
- Consistency Checker component applies consistency checking rules to enabled or disabled feature.
- Consistency Checker component displays a confirmation or rejection message to application engineer.
- Application engineer enters the values of parameterized variables.
- Application engineer saves customization information in the TargetSystemsFeatures table and the TargetSysVar table of the SPL model database.

The Feature Selector user interface of Figure 6-22 is used to locate and display a feature related artifacts (specifications, designs, source code files, etc.) from the SPL model. The stored path and name are used to locate the artifacts. The selected artifact is displayed using its original tool (Visio, Rational Rose, PowerPoint, MS Word, etc.).

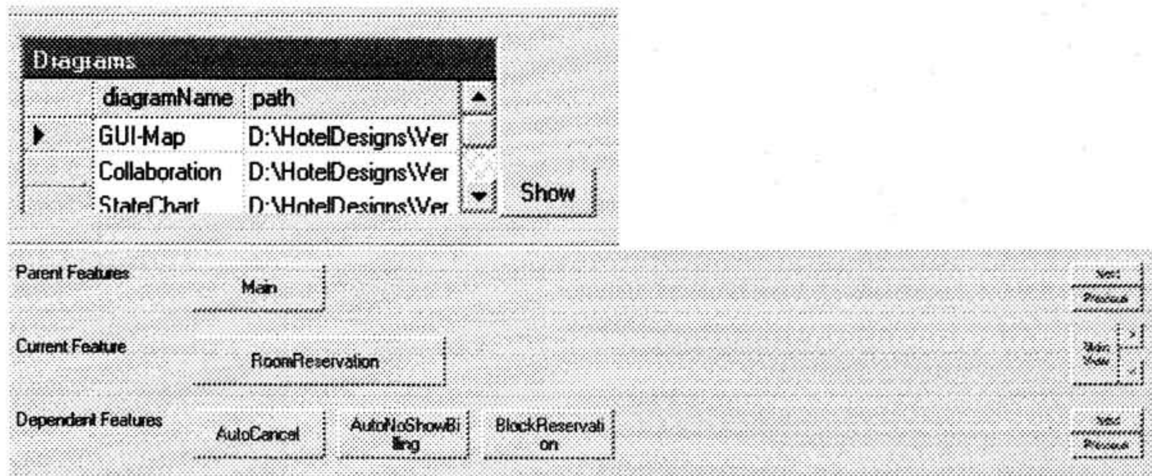


Figure 6-22 Feature Selector – diagrams

Figure 6-23 shows the sequence diagram for displaying feature related artifacts.

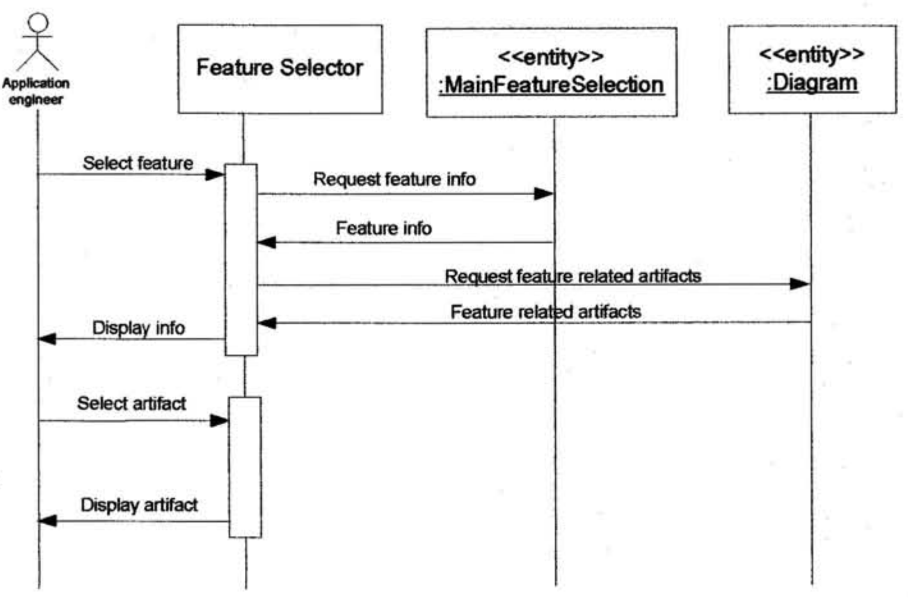


Figure 6-23 Display artifacts

The following explains the process:

- Application engineer selects a feature to view its artifacts.
- Feature Selector component requests feature information from the MainFeatureSelection table of the SPL model database.
- Feature Selector component requests related feature artifacts from the Diagram table of the SPL model database.
- Feature Selector component displays feature information and related feature artifacts.
- Application engineer selects a feature related artifact.
- Feature Selector displays selected artifact in its original tool (Visio, Word, etc.).

The Feature Selector user interface of Figure 6-24 is used to invoke a feature related web service method.

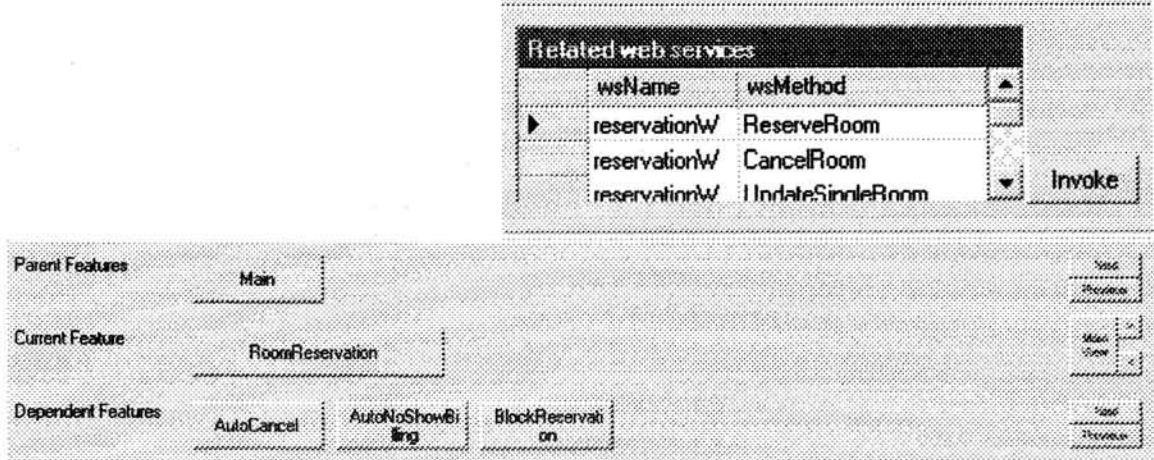


Figure 6-24 Feature Selector - related web services

Figure 6-25 shows the sequence diagram for invoking a web service method.

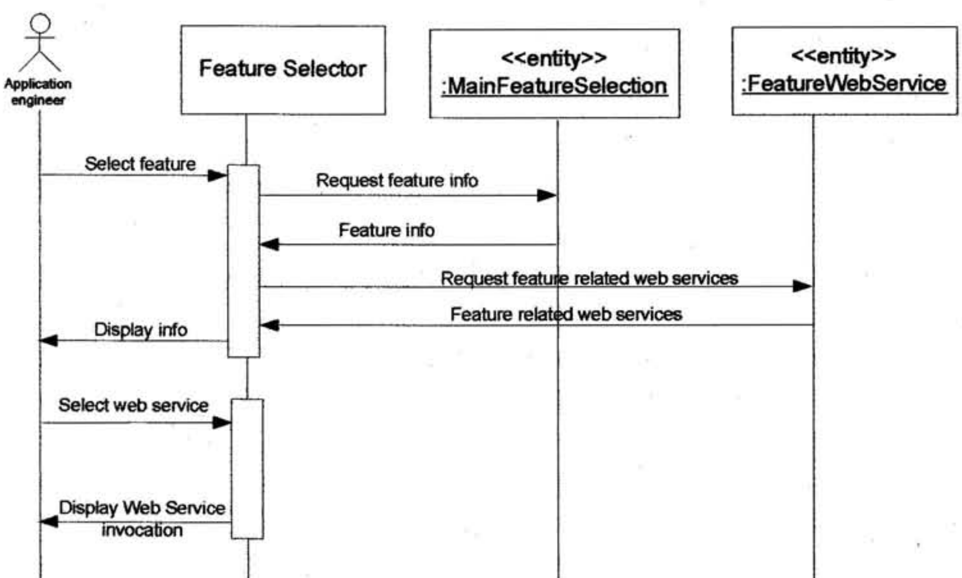


Figure 6-25 Web Service Invocation

The following explains the process:

- Application engineer selects a feature to invoke its related web services.
- Feature Selector component requests feature information from the MainFeatureSelection table of the SPL model database.
- Feature Selector component requests related feature web services from the FeatureWebService table of the SPL model database.
- Feature Selector component displays feature information and related feature web services.
- Application engineer selects a feature related web service.
- Feature Selector invokes the web service.

Figure 6-26 shows a sample web service method. The reserveRoom web service method is shown with all required input using the standard input interface that is provided with the .NET framework.

reservationWS

Click here for a complete list of operations.

reserveRoom

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
pName:	
pAddress1:	
pAddress2:	
pTel:	
pCreditCardNo:	
pExpirationDateStr:	
pCreditType:	
pResRoomType:	
pArrivalDateStr:	
pDaysNo:	
pNumberOcc:	

Invoke

Figure 6-26 Web Service invocation - ReserveRoom

Figure 6-27 shows a sample SOAP request and response for the reserveRoom web service method. This figure is displayed along with the input interface of Figure 6-26. It

describes the expected input and output types and location of the web service in XML/SOAP format.

```
POST /HotelSys/reservation/reservationWS.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/reserveRoom"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org
  <soap:Body>
    <reserveRoom xmlns="http://tempuri.org/">
      <pName>string</pName>
      <pAddress1>string</pAddress1>
      <pAddress2>string</pAddress2>
      <pTel>string</pTel>
      <pCreditCardNo>int</pCreditCardNo>
      <pExpirationDateStr>string</pExpirationDateStr>
      <pCreditType>string</pCreditType>
      <pResRoomType>string</pResRoomType>
      <pArrivalDateStr>string</pArrivalDateStr>
      <pDaysNo>int</pDaysNo>
      <pNumberOcc>int</pNumberOcc>
    </reserveRoom>
  </soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <soap:Body>
    <reserveRoomResponse xmlns="http://tempuri.org/">
      <reserveRoomResult>int</reserveRoomResult>
    </reserveRoomResponse>
  </soap:Body>
</soap:Envelope>
```

HTTP POST

The following is a sample HTTP POST request and response. The **placeholders** shown need to be replaced with actual values.

```
POST /HotelSys/reservation/reservationWS.asmx/reserveRoom HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

pName=string&pAddress1=string&pAddress2=string&pTel=string&pCreditCardNo=string&pExpirationDateStr=string
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<int xmlns="http://tempuri.org/">int</int>
```

Figure 6-27 SOAP message

Figure 6-28 shows the results returned from the reserveRoom web service method. The reservation number "1020" is returned in XML format. If the room reservation transaction is not successful, an integer value of zero is returned instead.

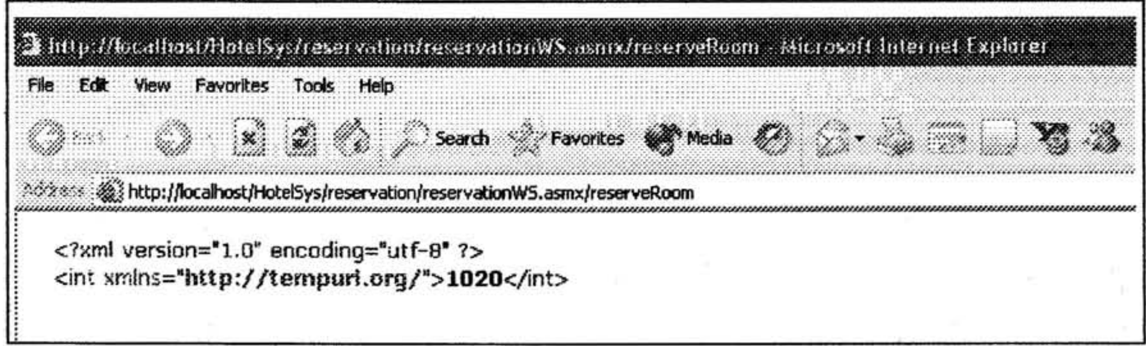


Figure 6-28 Results returned from roomReservation WS

6.2.2.2 Consistency Checker Component

This component is part of the Feature Selector component. It serves as a check for ensuring that features selected for the target application are consistent with each other. When a feature is selected using the "Update Selection" button of Figure 6-20, the Consistency Checker is invoked to verify selection. If a selected feature causes a violation to the SPL model, a warning message appears with proper explanation to the reason why this feature can not be selected or deselected. The following explains the consistency checking rules and action performed.

Consistency checking rules:

- **Rule 1:** A feature can not be deselected if it is a kernel feature.

  Action: Feature selection is disabled.

- **Rule 2:** Optional and alternative features can be selected or deselected if all of the following consistency rules are satisfied.

  Action: Based on rules 3 to 13.

- **Rule 3:** An optional feature cannot be selected if parent feature is an optional feature and it is not selected.

  Action: Message appears ("Feature can not be selected. Parent Feature must be selected first").

- **Rule 4:** An alternative feature cannot be selected if parent feature is an optional feature and it is not selected.

  Action: Message appears ("Feature can not be selected. Parent Feature must be selected first").

- **Rule 5:** An alternative feature can not be selected if parent feature is an alternative feature and it is not selected.

  Action: Message appears ("Feature can not be selected. Parent Feature must be selected first").

- **Rule 6:** An optional feature can not be selected if parent feature is an alternative feature and it is not selected.

  Action: Message appears ("Feature can not be selected. Parent Feature must be selected first").

- **Rule 7:** An optional feature can be selected if parent feature is kernel.

  Action: Feature is selected.

- **Rule 8:** An alternative feature can be selected if parent feature is kernel.

Action: Feature is selected and all of the other alternative features in the related set are deselected.

- **Rule 9:** An optional feature can not be deselected if it has a selected dependent feature.

  Action: Message appears ("Feature cannot be deselected. Dependent Features must be deselected first").

- **Rule 10:** An alternative feature cannot be deselected if it has a selected dependent feature.

  Action: Message appears ("Feature cannot be deselected. Dependent Features must be deselected first").

- **Rule 11:** An alternative feature cannot be selected if one of the other alternatives in the set of related alternatives is selected. Alternatives are mutually exclusive.

  Action: Only one of the set of related alternative features is selected. The other alternative features will be set to false.

6.2.2.3 Customization File Generator component

This component is responsible for generating a customization file automatically for each target application. The customization file is required for the customization process of client applications, described in Chapter 5 section 5.2, 5.4, and 5.5 (DCAC, DCAC-SC, and SCAC patterns). The three customization methods depend on this file. This component relies on the Feature Selector component, which sets feature selection status

to true/false and stores values of parameterized variables in the TargetSystemsFeatures table and the TargetSystemVar table of the SPLmodel database.

Figure 6-29 shows the graphical user interface for this component.

Figure 6-29 Customization File Generator component

Figure 6-30 shows the entity class diagram for the customization file. It consists of the following tables:

Feature: Stores all feature names and their selection status (Y/N).

Variable: Stores values of all parameterized variables grouped by feature name.

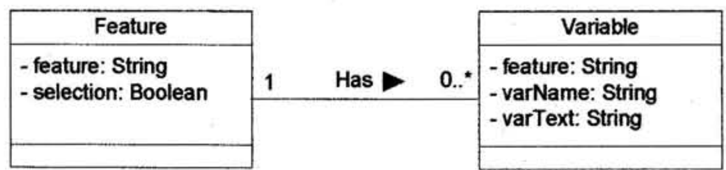


Figure 6-30 Entity Class Diagram - Customization File

Figure 6-31 shows the sequence diagram for generating a customization file for a target application.

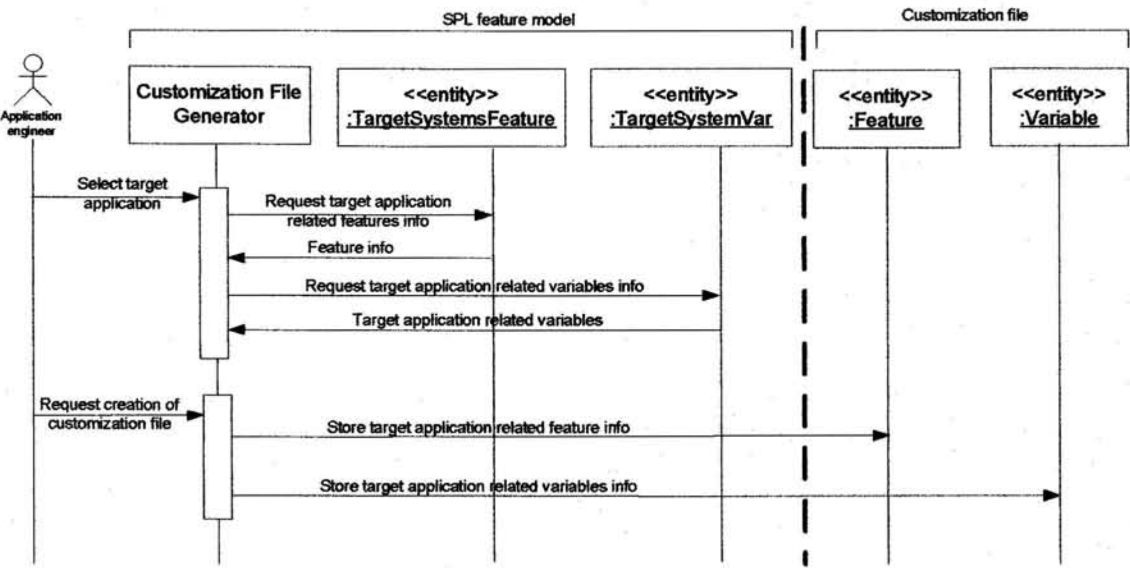


Figure 6-31 Customization File Generation

The following explains the process of generating a customization file for a specific target application:

- Application engineer selects a customized target application.

- Customization File Generator reads the related feature information of the selected target application from the TargetSystemsFeature table of the SPL model database.
- Customization File Generator reads the related parameterized variable information of the selected target application from the TargetSystemVar table of the SPL model database.
- Application engineer requests the creation of the customization file.
- Customization File Generator inserts feature names and their selection status of selected target application into the Feature table of the customization file.
- Customization File Generator inserts parameterized variables information of selected target application into the Variable table of the customization file.

6.2.3 Separation of concerns and source code integration subsystem

This subsystem is used to establish separation of concerns and integrate variable source code with kernel source code. It consists of three components: Variable Source Code Editor, Code Tracker, and Code Weaver. Figure 6-32 shows the subsystem and the interaction between its components and related files. The following sections describe in detail each component in the subsystem.

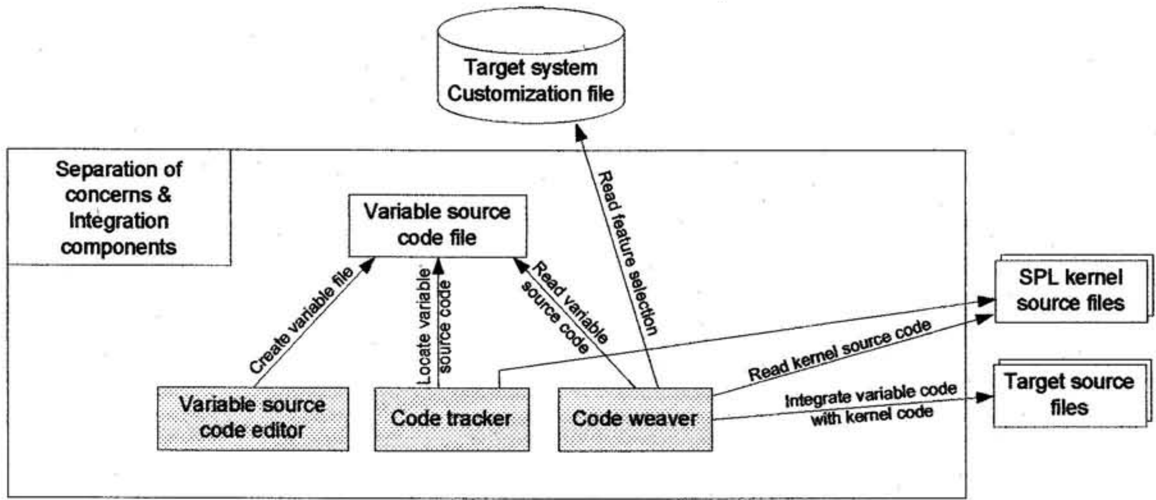


Figure 6-32 SPLET – Separation of Concerns & Code Weaving

6.2.3.1 Variable Source Code Editor Component

This component is used to relate optional and alternative source code to features for the purpose of establishing separation of concerns between variable source code and kernel source code. Each feature in the feature tree has a related variable source code file that is created and manipulated internally using the SPLET prototype. Features that interact with other features have separate variable files. The individual variable files are composed automatically into a single variable source code file that is used by the code weaver component to integrate variable source code with kernel source code according to the customization methods, described in the DCAC-SC and SCAC patterns in Chapter 5. The file structuring, manipulation, and the composition of variable source code files into one variable file are done internally in SPLET. Users interact with the user friendly interfaces to create all variable source code.

This component consists of three major functions:

- Single-feature source code generation: This function is used for relating variable source code to a single feature. Each feature has one file that stores all related variable source code. Each block of variable source code in this file is identified by an insertion point name.
- Multi-feature source code generation: This function is used when a feature has to interact with other features. Every set of interacting features has one file that stores all related variable source code. Each block of variable source code in this file is identified by an insertion point name.
- Composed variable file generation: This function is used to compose all individual files into a single variable file. The code weaver component reads this file in the integration process.

The following sections explain each function in detail.

6.2.3.1.1 Single feature source code generation

Figure 6-33 shows the graphical user interface used to create variable source code for each feature.

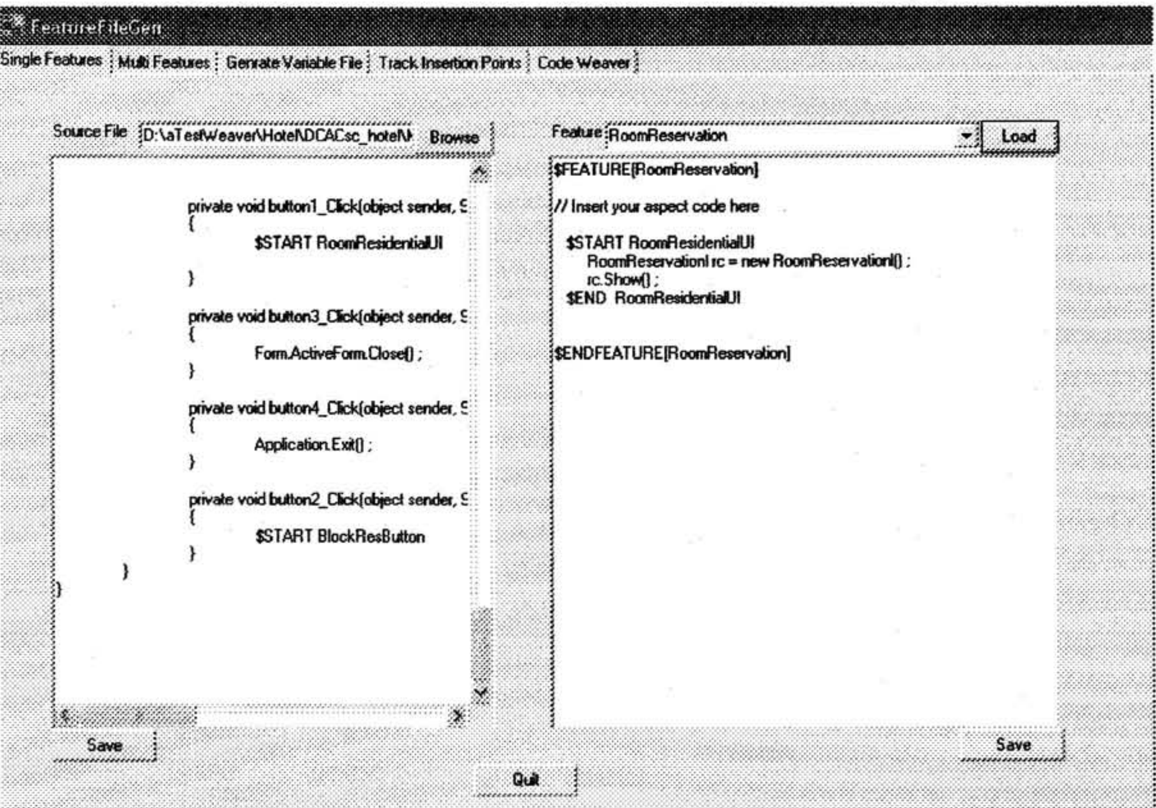


Figure 6-33 Variable Source Code Editor - Single Features

Figure 6-34 shows the sequence diagram for creating feature related variable source code.

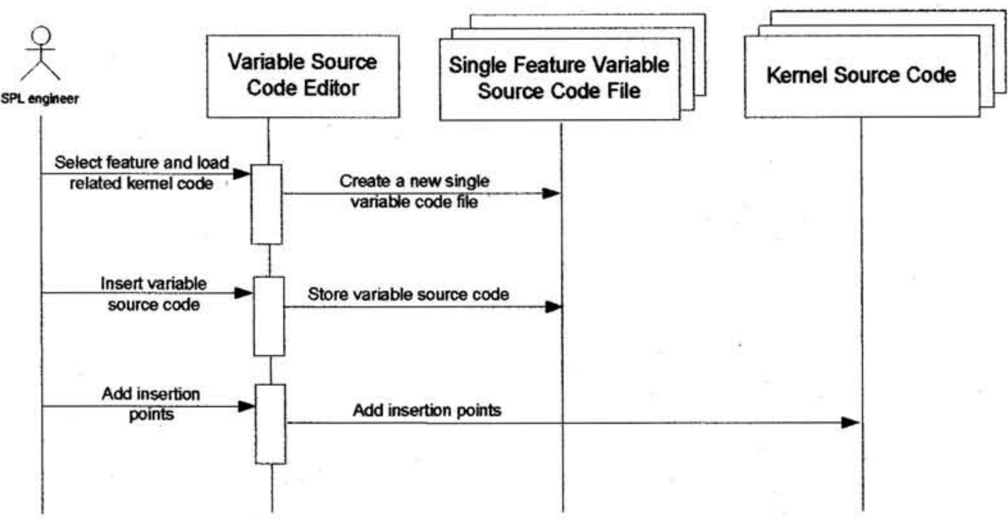


Figure 6-34 Single Feature Variable Source Code File Creation

The following explains the process:

- SPL engineer selects a feature to add its variable source code.
- Variable Source Code Editor component creates a new single feature variable source code file or loads an existing file.
- Variable Source Code Editor component creates a predefined template in the single feature variable source code user interface with the appropriate SPLET commands. SPLET commands are described in detail in Chapter 5 sections 5.4 and 5.5 (DCAC-SC and SCAC patterns). The following is a sample template for BlockCheckin feature:

```
$FEATURE[BlockCheckin]
$START Insertion_Point_Name
    // insert variable source code here
$END Insertion_Point_Name
$ENDFEATURE[BlockCheckin]
```

- The Insertion_Point_Name is replaced with the actual insertion name.
- Variable source code is inserted after the $START command line.
- In the kernel source code, the insertion point name is inserted at the location where the variable source code is expected to be merged in the integration process.

- The code weaver component decides on which optional or alternative source code is expected to be integrated with the kernel source code, described later in the Code Weaver component.

6.2.3.1.2 Multi feature source code generation

Figure 6-35 shows the graphical user interface used to create variable source code for each feature.

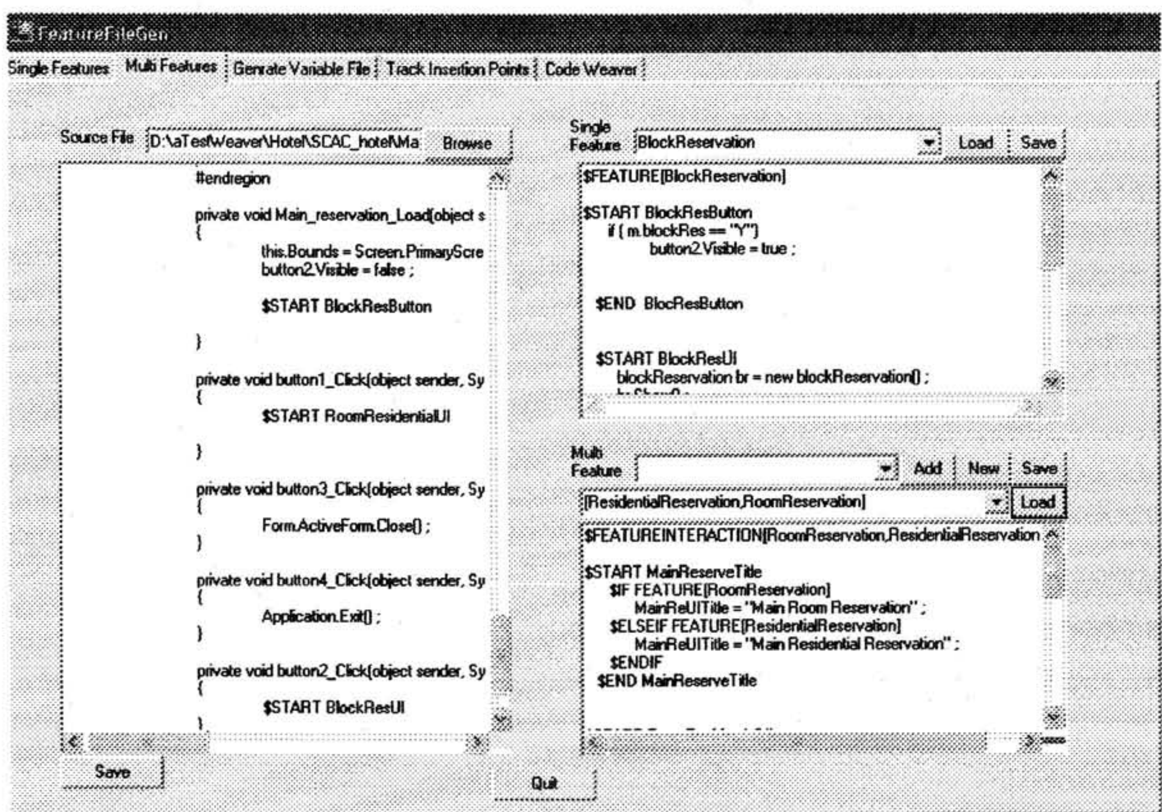


Figure 6-35 Variable Source Code Editor - Multi Features

Figure 6-36 shows the sequence diagram for creating multi features related variable source code.

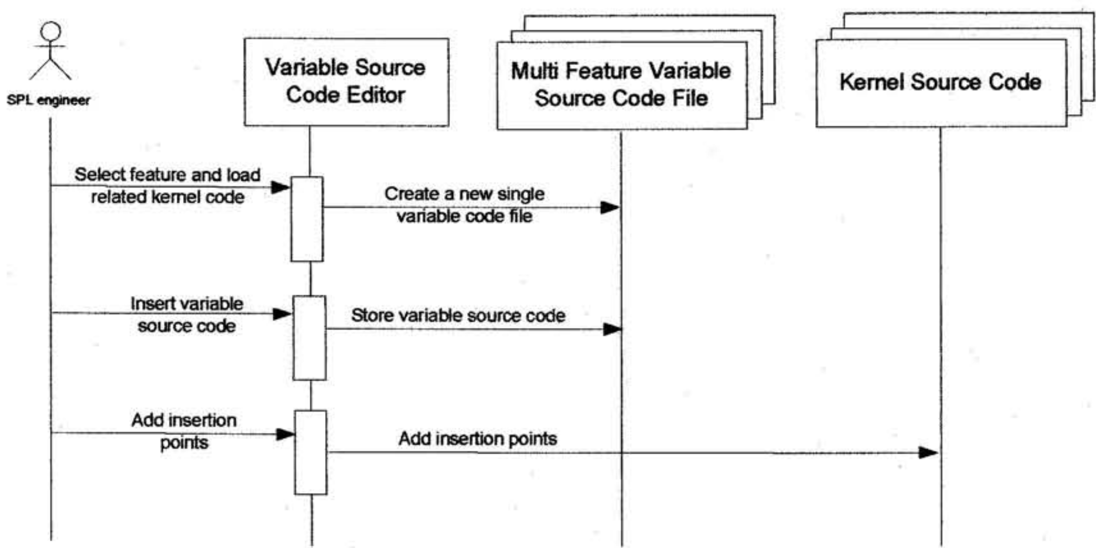


Figure 6-36 Multi Feature Variable Source Code File Editor

The process in figure 6-36 is the same as the single feature source code generation process, described in section 6.2.3.1. However, the predefined template in the multi feature variable source code user interface uses different SPLET commands. SPLET commands are described in detail in Chapter 5 sections 5.4 and 5.5 (DCAC-SC and SCAC patterns). The following is a sample template for RoomReservation and ResidentialReservation features:

```
$FEATUREINTERACTION[RoomReservation, ResidentialReservation]
  $START Insertion_Point_Name

        // insert variable source code here

  $END Insertion_Point_Name
$ENDFEATUREINTERACTION[RoomReservation, ResidentialReservation]
```

6.2.3.1.3 Composed variable source code file generation

Figure 6-37 shows the graphical user interface used to create a composed source code variable file from all the individual variable source code files. The generated variable source code file is used by the Code Weaver component to integrate variable source code with kernel source code.

Figure 6-37 Variable Source Code Editor - Composed Features

Figure 6-38 shows the sequence diagram for creating the variable source code file.

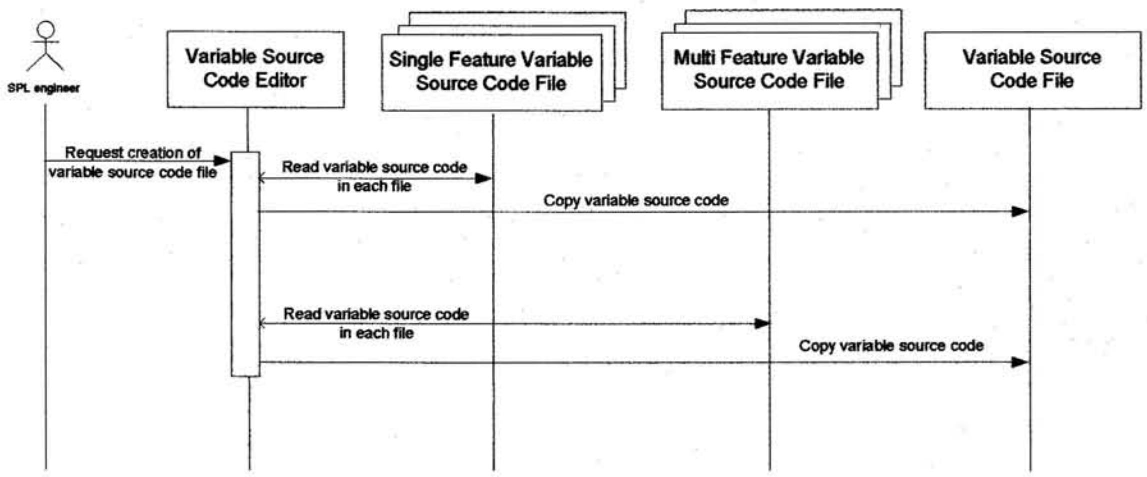


Figure 6-38 Creation of Variable Source Code File

The following explains the process:

- SPL engineer requests from the Variable Source Code Editor component to generate a variable source code file.
- Variable Source Code Editor component reads variable source code in each Single Feature Variable Source Code File and copies the read variable source code into the composed Variable Source Code File.
- Variable Source Code Editor component reads variable source code in each Multi Feature Variable Source Code File and copies the read variable source code into the composed Variable Source Code File.
- A composed variable source code file is generated.

6.2.3.2 Code Tracker

Insertion point names can grow large in number and need a facility to locate them in both the variable files and their corresponding insertion point names in the kernel source code. The Code Tracker component in Figure 6-39 is used for this purpose. The code Tracker component can track insertion point names in two ways:

- **Feature tracking:** Tracks all insertion point names by feature name or interacting features. The dropdown list of features contains all single feature names and multi feature names. Multi feature names represent interacting features. They are combined between two brackets. For example, RoomReservation feature and ResidentialReservation feature are two interacting features that are created with the Variable Source Code Editor component. The dropdown list shows these two interacting features as:

  [RoomReservation, ResidentialReservation]

- **Insertion name tracking:** Tracks a specific insertion point name in all kernel source code files and shows its related feature or interacting features.

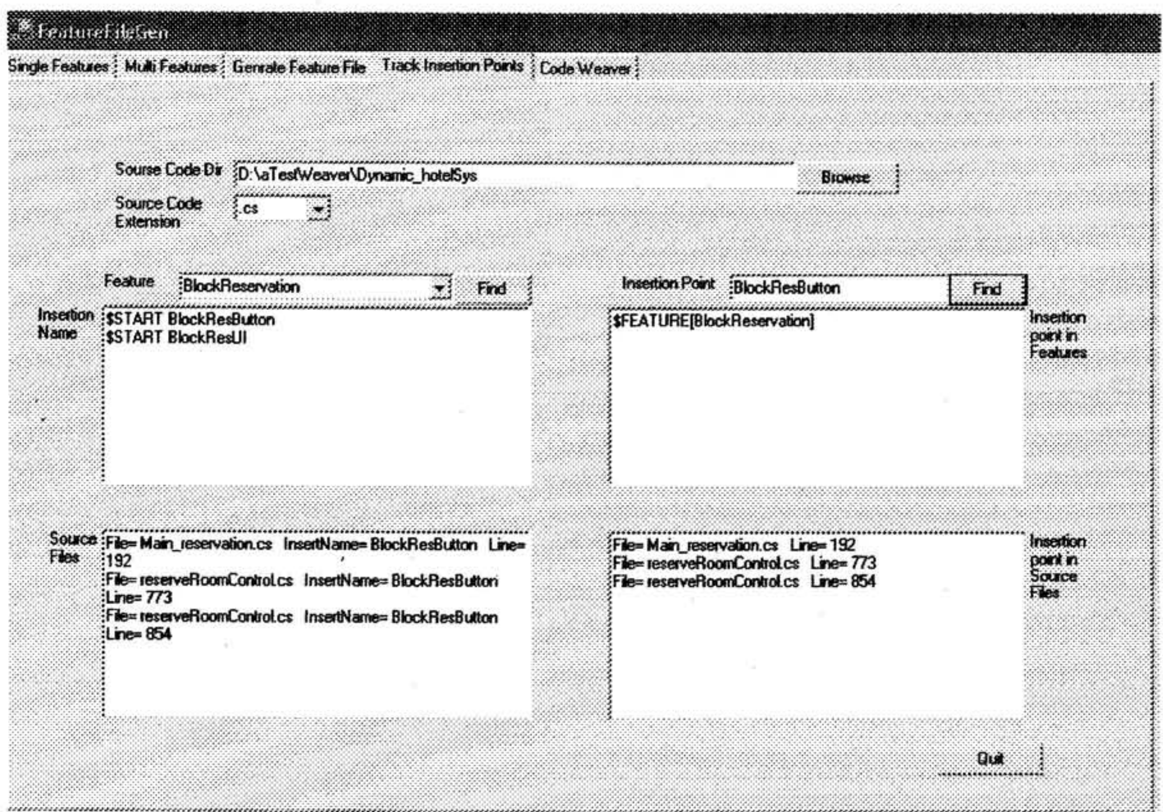


Figure 6-39 Code Tracker

Feature tracking:

Figure 6-40 shows the sequence diagram for tracking feature related insertion points and their location in the kernel source code files.

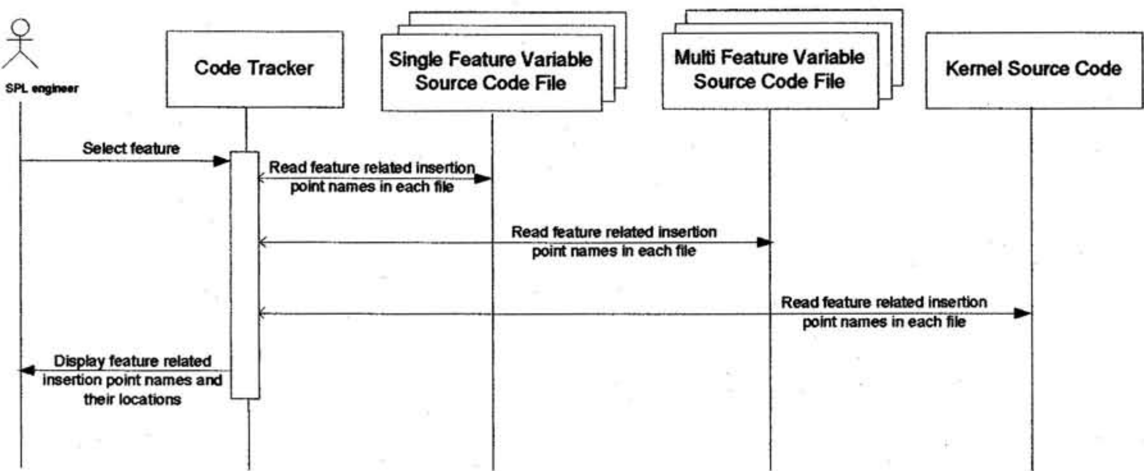


Figure 6-40 Tracking of feature related insertion points

The following explains the process:

- SPL engineer selects a feature or a set of interacting features.
- Code Tracker component reads feature related insertion point names in the Single Feature Variable Source Code file.
- Code Tracker component reads feature related insertion point names in the Multi Feature Variable Source Code file.
- Code Tracker component reads feature related insertion point names in the kernel source code files.
- Code Tracker component displays all feature related insertion point names.

- Code Tracker component displays all kernel source code files that contain each insertion point name and the location (line number) of each insertion point in the kernel source code file.

Insertion name tracking:

Figure 6-41 shows the sequence diagram for tracking a specific insertion point name in the kernel source code files and its related feature or interacting features.

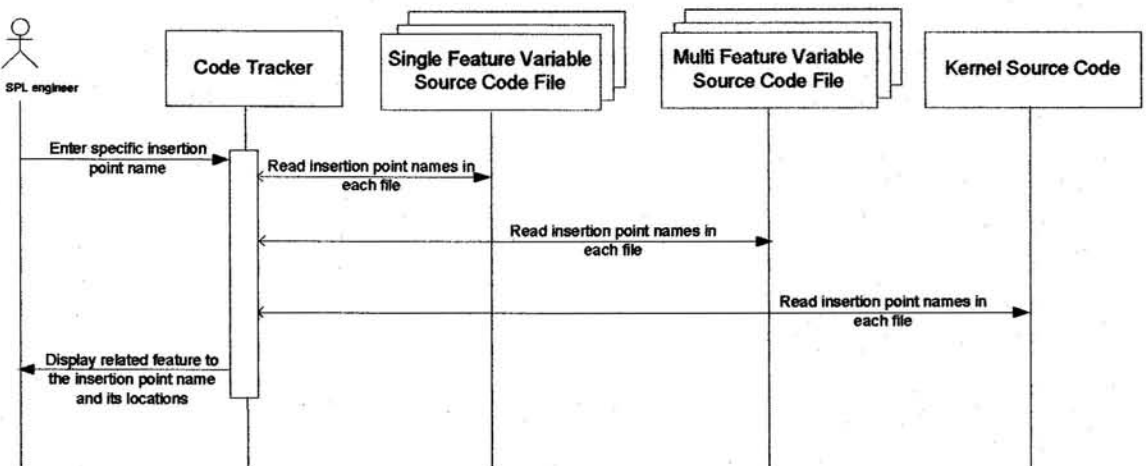


Figure 6-41 Tracking of specific insertion point name

The following explains the process:

- SPL engineer enters a specific interaction point name.
- Code Tracker component reads the insertion point name in all Single Feature Variable Source Code files.
- Code Tracker component reads the insertion point name in all Multi Feature Variable Source Code files.

- Code Tracker component reads the insertion point name in all kernel source code files.
- Code Tracker component displays the insertion point name related feature or interacting features.
- Code Tracker component displays all kernel source code files that contain the insertion point name and its location (line number) in the kernel source code files.

The tracked information is used to locate variable source code in their corresponding variable source code files and kernel source code. This information helps users to make necessary modifications and updates to the SPL application using the Variable Source Code Editor component.

6.2.3.3 Code Weaver

The Code Weaver is built to support the integration process that is described in the Dynamic Client Application Customization with Separation of Concerns (DCAC-SC) and the Static Client Application Customization (SCAC) approaches in Chapter 5 sections 5.4 and 5.5. It is responsible for integrating kernel source code with optional and alternative source code using the automatically composed variable source code file and feature selection. Figure 6-42 shows the main user interface of the Code Weaver component.

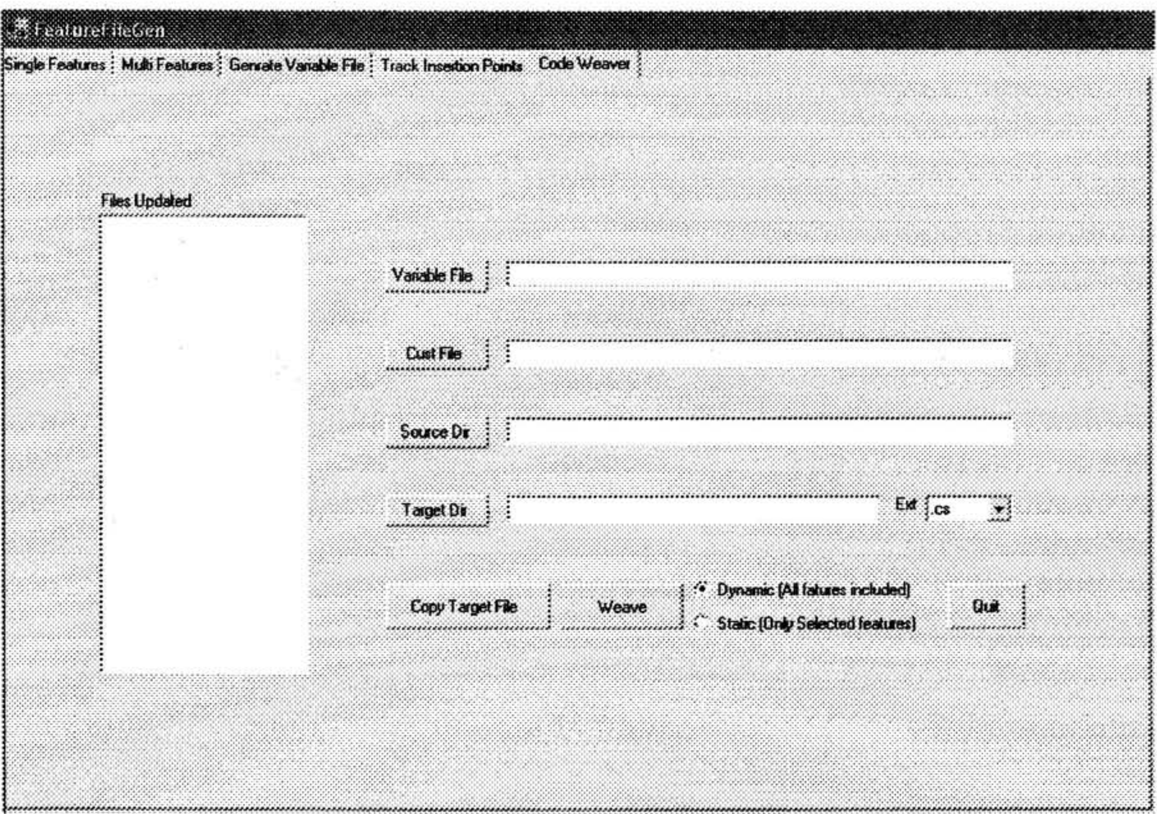


Figure 6-42 Code Weaver

For the dynamic approach of DCAC-SC, Figure 6-43 shows the overall integration process. In this integration method, all optional and alternative source code from the variable source code file is integrated with kernel source code. The integrated source code is then compiled to produce a SPL application that can be dynamically customized at run time. The integrated SPL application is customized after the code weaving and compilation processes. The Feature Selector, Consistency Checker, and Customization File Generator components under the Customization subsystem are used to customize target applications, described in section 6.2.2.1, 6.2.2.2, and 6.2.2.3.

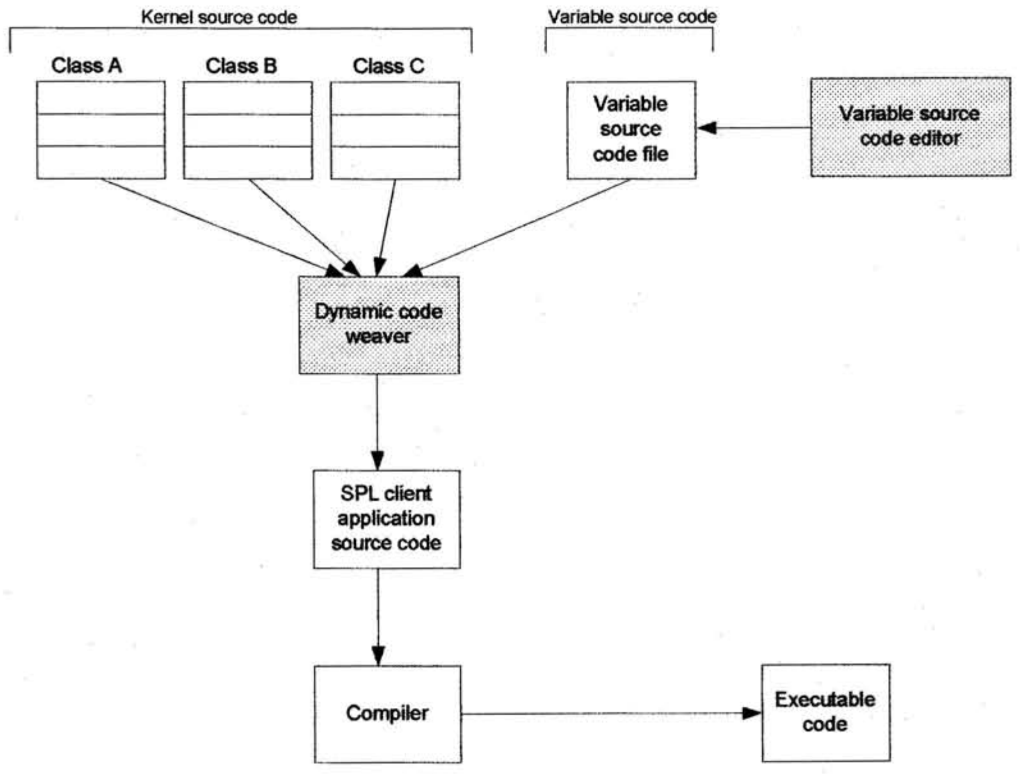


Figure 6-43 Dynamic integration

Figure 6-44 shows the sequence diagram for the integration process of kernel source code with variable source code using the Code Weaver component with the DCAC-SC approach.

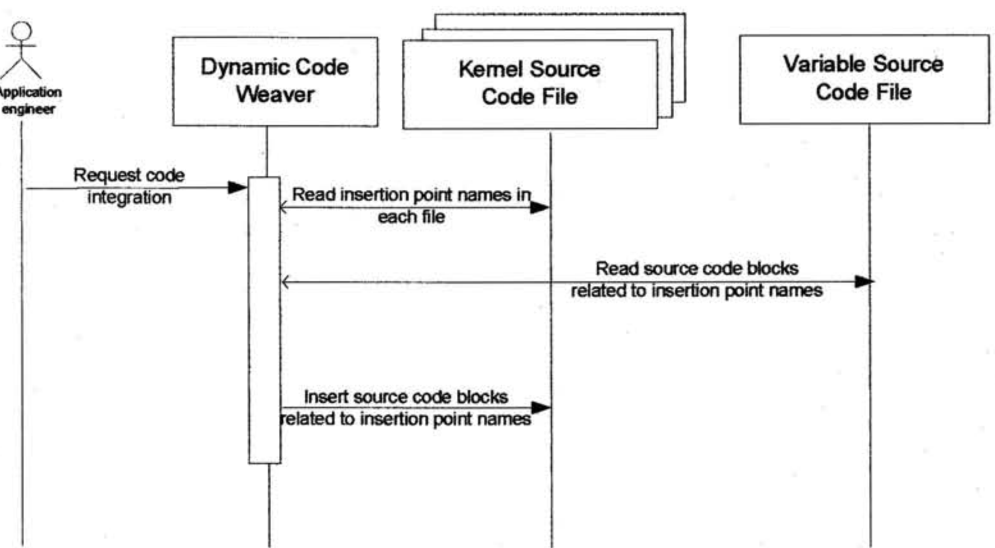


Figure 6-44 Code Weaving for DCAC-SC Method

The following explains the process:

- Application engineer requests integration of kernel source code with variable source code using the dynamic method of integration (DCAC-SC method).
- Dynamic Code Weaver component reads insertion point names in kernel source code files, which correspond to join points in Aspect Oriented Programming.
- When an insertion point is located in the kernel source code file, the dynamic Code Weaver component reads the variable source code file to locate the corresponding insertion point name.

- The variable source code block that is related to the found insertion point name is integrated with kernel source code at the specified location in the kernel source code.

For the static approach of SCAC, Figure 6-45 shows the overall integration process. In this integration method, only selected optional and alternative feature related source code blocks are integrated with kernel source code. Therefore, in order for the dynamic code weaver component to perform the integration, the customization process of selecting desired features is performed before the integration process. The Feature Selector, Consistency Checker, and Customization File Generator components under the Customization subsystem are used to generate a customization file of selected features to be used by the dynamic code weaver to make decisions on which variable source code blocks to include or ignore. After the integration process is complete, the integrated source code is the compiled to produce an executable customized target application.

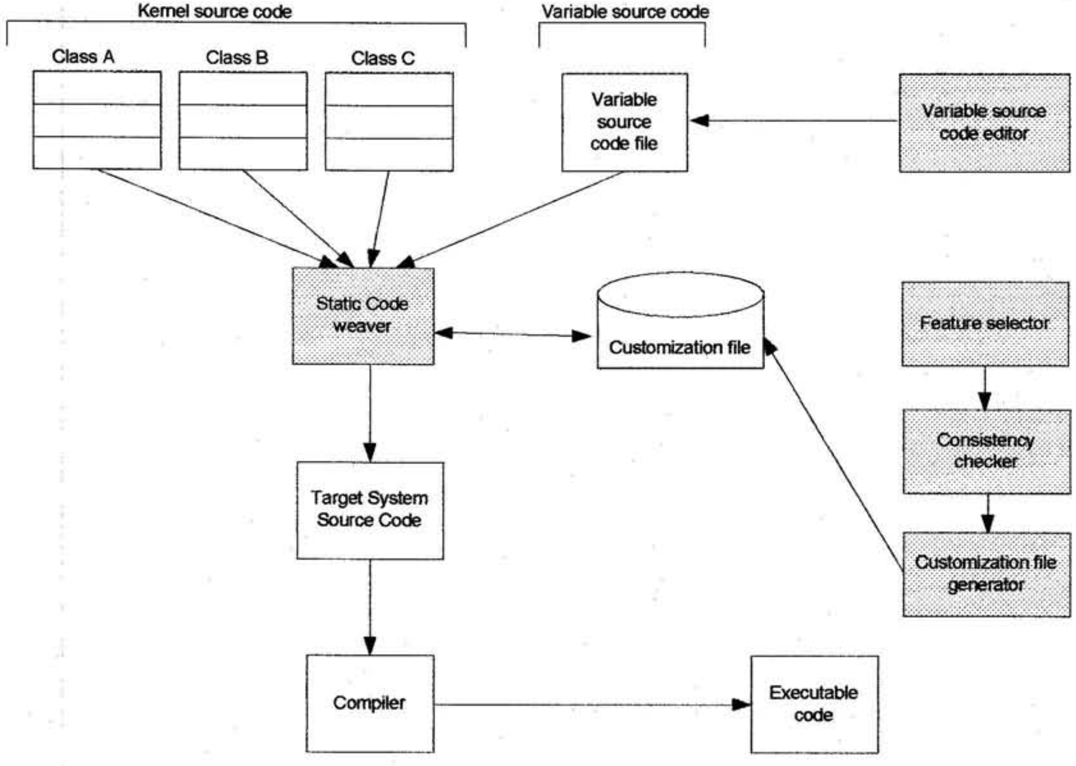


Figure 6-45 Static integration

Figure 6-46 shows the sequence diagram for the integration process of kernel source code with variable source code.

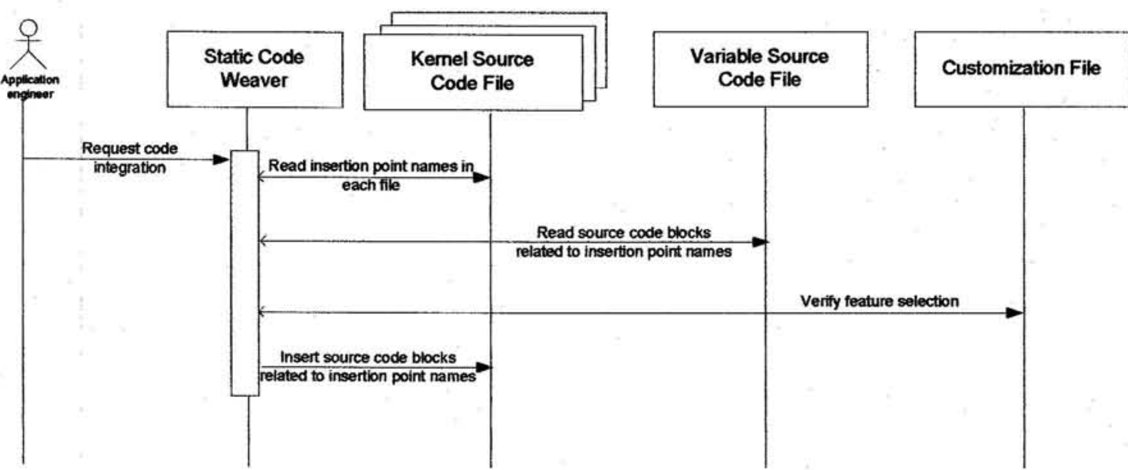


Figure 6-46 Code Weaving for SCAC Method

The following explains the process:

- Application engineer requests integration of kernel source code with variable source code using the static method of integration (SCAC method).
- Static Code Weaver component reads insertion point names in kernel source code files.
- When an insertion point is located in the kernel source code file, the Code Weaver reads the variable source code file to locate the corresponding insertion point name.
- Static Code Weaver component consults the customization file to verify feature selection.
- If the feature is selected, the Static Code Weaver integrates variable source code block with kernel source code at the specified insertion location in the kernel source code.
- If the feature is not selected, the Static Code Weaver ignores the integration of variable source code block with kernel source code.

Figure 6-47 shows two sample variable files that are generated by the Variable Source Code Editor component. The Code Weaver component reads the generated variable file and applies the selected integration method in the integration process.

Static feature file

```
$FEATUREINTERACTION[RoomReservation,ResidentialReservation]

  $START MainReserveTitle
    $IF FEATURE[RoomReservation]
        MainReUITitle = "Main Room Reservation" ;
    $ELSEIF FEATURE[ResidentialReservation]
        MainReUITitle = "Main Residential Reservation" ;
    $ENDIF
  $END MainReserveTitle

//////////////

  $START RoomResidentialUI

    $IF FEATURE[RoomReservation]                    // Alternative feature
      RoomReservationI rc = new RoomReservationI() ;
      rc.Show() ;

    $ELSEIF FEATURE[ResidentialReservation]    // Alternative feature
      ResidentialReservationI rs = new ResidentialReservationI() ;
      rs.Show() ;
    $ENDIF

  $END  RoomResidentialUI

$ENDFEATUREINTERACTION[RoomReservation,ResidentialReservation]

////////////////////////////////////////////////////////////////////////////////
$FEATURE[BlockReservation]                    // Optional Feature

  $START BlockResButton
     //create Block reservation button
     blockRes_button.visible = true;

  $END  BlocResButton


  $START BlockResUI
     blockReservation br = new blockReservation() ;
     br.Show() ;
  $END BlockResUI

$ENDFEATURE[BlockReservation]
```

Dynamic feature file

```
$FEATUREINTERACTION [RoomReservation,ResidentialReservation]

  $START MainReserveTitle

      if( roomRes == "y" )
            MainReUITitle = "Main Room Reservation" ;

      elseif ( residRes == "y")
            MainReUITitle = "Main Residential Reservation" ;

  $END MainReserveTitle

//////////////////////////
  $START RoomResidentialUI

     if( roomRes == "Y")
     {
        // Display RoomReservationUI
        RoomReservationI rc = new RoomReservationI() ;
        rc.Show() ;
     }
     else if( residRes == "Y")
     {
        // Display ResidentialReservationUI
        ResidentialReservationI rs = new ResidentialReservationI() ;
        rs.Show() ;
     }
  $END  RoomResidentialUI

$ENDFEATUREINTERACTION [RoomReservation,ResidentialReservation]

////////////////////////////////////////////////////////////////////////////////
$FEATURE[BlockReservation]

  $START BlockResButton
    if (blockRes == "Y")
    {
        // Create block reservation button
        blockRes_button.visible = true;
    }
  $END  BlocResButton


  $START BlockResUI
    if (blockRes == "Y")
    {
        blockReservation br = new blockReservation() ;
        br.Show() ;
    }
  $END BlockResUI

$ENDFEATURE[BlockReservation]
```

Figure 6-47 Samples of Variable File

6.2.4 Utility subsystem

The utility subsystem consists of the File Extractor component. It is used as a supporting tool for retrieving analysis, designs, source code files, and test procedures for target applications. Figure 6-48 shows the Utility subsystem.

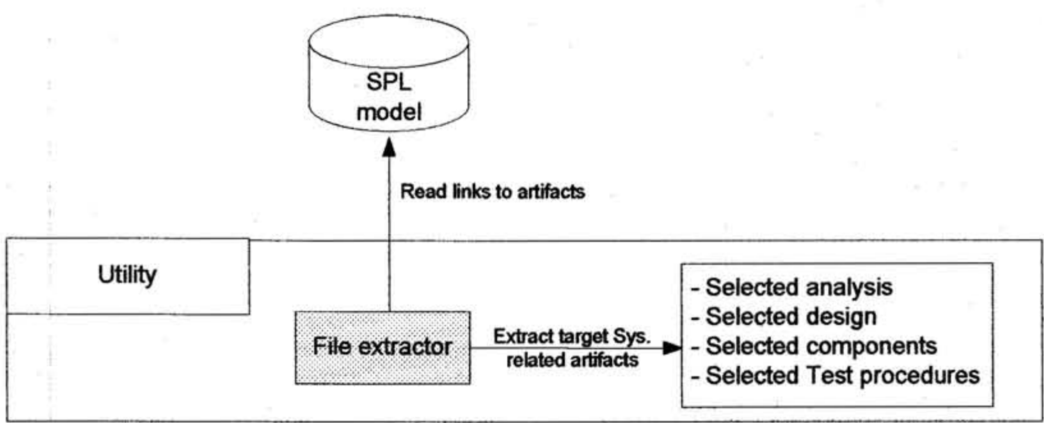


Figure 6-48 SPLET - Utility

Figure 6-49 shows the main user interface of the File Extractor component.

Print

Print / Files Extraction

Target System Name | Load

Destination | Browse

Specifications | Source Code
Analysis | Tests
Design | Other | All

Extract Selected Files | Quit

Figure 6-49 File Extractor utility

File extraction is based on the Feature Editor and Feature Selector components. The Feature Editor allows users to enter links to files/diagrams that are related to each feature in the Diagram table of the SPL model database. Files are categorized in the Diagram table as: specifications, analysis, design, source code, tests, and other. The File Extractor utility creates sub-directories in the destination path for each category type. It then copies files of checked category types of selected target application in their related sub-directories. The File Extractor utility consults the TargetSystemsFeature table in the SPL model database to verify feature selection. Only files related to selected features of a target application are copied into their corresponding directory.

6.3 Validation

This research is validated through two case studies using a product line independent proof-of-concept prototype (SPLET). The two case studies apply the software product line service-oriented development approach to the design, development and customization of the proposed architecture and implementation. The two case studies are:

- Hotel Product Line
- Radio Frequency Management Product Line

6.3.1 Validation process

a) Developed a product line proof-of-concept prototype to (SPLET), which supports the design, development, and customization of software product lines that are based on web services. The SPLET prototype covers the life cycle of software product lines

from the SPL engineering phase to the application engineering phase. The SPLET prototype was used as follow:

- In the SPL engineering phase, SPLET was used in the two case studies to create a SPL feature model and associate web service components and artifacts (specifications, design models, test files, source code) to their related features.
- In the application engineering phase, SPLET was used in the two case studies to select desired features, run consistency checking rules, and customize target applications using all of the three development approaches.
- The two case studies applied the proposed methods of separation of concerns and source code integration of variable source code and kernel source code according to the DCAC-SC and SCAC patterns.

b) Designed the two SPL case studies according to the proposed design approach. The design included the following multiple-view models:

- Use case modeling: Captures the overall software product line requirements.
- Feature modeling: A feature dependency model was derived from the use case model. The feature model was used to depict the kernel, optional, and alternative features in the SPL application.
- Entity class modeling: was used to depict the needed input when developing web services.

- User interface navigation modeling: Shows the navigation between kernel, optional, and alternative user interface objects.
- Interaction modeling: Describes the interaction between the user interfaces and web services.
- Activity modeling: Describes the workflow of each user interface object.
- Software architecture modeling: Identifies the required web services and their input and output.
- Component interface modeling: Objects from the interaction model are designed as components in terms of their interfaces and interconnections. User interface components communicate with web services and each other through ports, which support provided and/or required interfaces

c) Implemented three prototypes for each case study to validate the three development approaches described in this research. The two case studies were implemented according to:

- Dynamic client application customization (DCAC)
- Dynamic client application customization with separation of concerns (DCAC-SC).
- Static client application customization with separation of concerns (SCAC).

d) Derived target applications from the SPL architecture and components. Each SPL implementation of the two case studies was customized to derive two target applications.

e) Each derived target application was tested using conventional functional testing methods to verify the correct customization and execution of derived applications.

6.4 Summary

This chapter has described the Software Product Line Environment prototype (SPLET) as a proof of concept for this research. SPLET is designed to be a product line independent prototype that covers the product line life cycle, which includes the software product line engineering phase and the application engineering phase (SPL customization). SPLET is based on dividing a SPL application into features that are categorized as kernel, optional, and alternative. Features are the main driver for organizing SPL components and customizing target applications. The SPLET prototype helps in visualizing the overall SPL system by providing a flexible navigation facility through the SPL model, and provides the needed facilities to customize target applications. It consists of four subsystems: software product line environment, customization, separation of concerns, and supporting utility.

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

7. CONTRIBUTIONS AND FUTURE RESEARCH

7.1 Introduction

This dissertation has developed an approach for designing a Software Product Line (SPL) based on web services. It addressed the unique issues of using the web service technology in the designing approach. This research also described three software development environments to develop the proposed product line design and support the automatic customization of SPL architecture and components. The three approaches followed the same design architecture, but differed in how separation of concerns is used for software development and customization. Each development approach was implemented with specific consideration to one of the customization methods described in this research. The design, development, and customization methods were supported by a product line independent customization prototype to help developers and application engineers to create a customizable SPL application and generate target applications automatically from the reusable service-oriented product line.

7.2 Research Contribution

This research has focused in designing, developing, and customizing software product lines that are based on web services. A proof-of-concept prototype was developed to cover the software product line life cycle from the SPL engineering phase to the

application engineering phase. The main contributions of this research effort are as follows:

a) This research effort has developed a multiple-view modeling approach, which extends the Product Line UML-based Software Engineering environment (PLUS) to address the unique issues related to web services. The multiple-view model defines the different characteristics of a service-oriented software family, including the commonality and variability among the members of the family. In the design approach, several multiple-view models are created specifically for a software product line service-oriented architecture, including user interface navigation modeling, interaction modeling, activity modeling, software architecture modeling, and component interface modeling.

b) A major contribution of this research effort is the design of the three software development environments to support the automatic customization of SPL service-oriented architecture and components. The three approaches are:

    - Dynamic client application customization (DCAC): This approach provides an automated customization method of target applications at system run time. Product lines are automatically customized by selecting desired features and entering values of parameterized variables to satisfy the execution of a specific target application. Selected features and parameters are stored in a customization file that is used by the target application objects to customize the client application user interfaces and their workflows at system run time.

The benefits of reuse can be achieved by deriving many target applications from the customizable SPL application without the need to modify any of the source code.

- Dynamic client application customization with separation of concerns (DCAC-SC): The second development approach is an extension to the first method (DCAC) to include separation of concerns, where optional and alternative source code is separated from kernel source code into a variable source code file. During source code integration, the variable source code file is used to integrate kernel source code with optional and alternative source code. The result of the integration process is a combined set of source code for the entire software product line, including *all* optional and alternative source code. The source code integration process and compilation are performed only once to generate a customizable SPL application at system run time. Target applications will rely on the dynamic client application customization, which is identical to that produced by the first approach (DCAC).

Separation of concerns is used to reduce complexity of developing software product lines and improve system maintenance by uniquely identifying variable source code and kernel source code. Variable source code can be manipulated separately within the SPL environment then automatically integrated with kernel source code.

- Static customization of client application with separation of concerns (SCAC): This approach is based on static customization of application objects at system derivation time. Client objects are customized by integrating kernel source code with *only* the selected optional and alternative source code from the variable source code file. With this approach, there is no customization at system run time. Using the static customization approach, the target application's source code is derived automatically from the SPL architecture and components. This approach is suitable for SPL applications that require distribution of only needed target application source code.

c) Another major contribution of this research is the development of a product line independent proof-of-concept prototype (SPLET), which supports the design, development, and customization of software product lines that are based on web services. SPLET covers the entire life cycle of software product lines from the SPL engineering phase to the application engineering phase.

- In the SPL engineering phase, SPLET enables SPL engineers to create a SPL feature model and associates web service components and artifacts (specifications, design models, test files, source code) to their related features.
- In the application engineering phase, SPLET provides facilities that enables application engineers to select desired features, run consistency checking rules, and customize target applications using one of the three development approaches.

- Provide the necessary tools to establish separation of concerns between variable source code and kernel source code with the facility to integrate them according to one of the integration methods described in the DCAC-SC and SCAC patterns.

d) Another contribution of this research is the development of a feature-based description language for separation of concerns. This language is used to identify optional and alternative source code for creating the variable source code file, which is used by the code weaver component in the separation of concerns subsystem of the SPLET prototype to integrate kernel source code with variable source code according to the dynamic integration method of DCAC-SC or the static method of SCAC.

e) Another contribution of this research is the development of a code integration engine (code weaver component in SPLT) that is able to interpret the developed feature-based description language to integrate kernel source code with variable source code according to the dynamic or static customization methods.

7.3 Future Research

This section describes possible future research in the area of software product line based on web services.

7.3.1 Testing of software product lines

This research has developed a method for integrating variable source code with kernel source code to produce a SPL application that is configured for dynamic customization at run time or a target application that is customized statically at source code integration time. There is a need to generate feature related test procedures to verify the integration and customization of target applications.

7.3.2 Transaction of web services using customizable workflows

This research focused on designing, developing, and customizing web service-based SPL applications. A possible future work in this area can be conducted to ensure that customizable workflows produce a successful business transaction, especially keeping track of transactions that require different inputs to and outputs from several loosely-coupled web service components.

7.3.3 Performance of SPL applications based on web services

This research has introduced three development approaches for software product lines based on web services. Performance issues were not addressed in this research. A possible future work can be pursued to measure performance level and reliability of using web services in a customizable service-oriented architecture. Also, Performance

measurement and reliability of the Internet usage of web services can be compared with other component-based applications using CORBA or DCOM middleware.

7.3.4 Evolution of SPL applications based on web services

The issue of evolution of SPLs based on web services is not addressed in this research effort. A study can be conducted to investigate the different possibilities to evolve the customizable SPL system with minimum change to the original source code.

7.4 Summary

This dissertation has focused on designing, developing, and customizing web service-based SPL applications. This research addressed the unique issues of using web services in the designing approach. It also described three different development approaches to develop the proposed product line design. A domain independent proof-of-concept prototype was developed to support the ideas presented in this research. The design, development, and customization approaches were applied to two case studies: Hotel System and Radio Frequency Management System to validate this research. The contributions of this research effort were described in this chapter.

العنوان: Software Product Line Engineering Based on Web Services
المؤلف الرئيسـي: Saleh, Mazen M. Aquil
مؤلفين آخرين: Gomaa, Hassan(Super.)
التاريخ الميلادي: 2005
موقع: فيرفاكس، فرجينيا
رقم MD: 618453
نوع المحتوى: رسائل جامعية
اللغة: English
الدرجة العلمية: رسالة دكتوراه
الجامعة: George Mason University
الكلية: Volgenau School of Engineering
الدولة: الولايات المتحدة الأمريكية
قواعد المعلومات: Dissertations
مواضيع: البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط: https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

TABLE OF CONTENTS

Page

ABSTRACT XI

1. INTRODUCTION 1

1.1 Background 1
1.2 Research Problem and Approach 2
1.3 Importance and Rationale of This Research 3
1.4 Terminology 3
1.5 Organization 5

2. RELATED WORK 6

2.1 Introduction 6
2.2 Software Product Lines 6
2.3 Evolutionary Software Product Line Engineering Process 7
2.4 Multiple-View Models of Software Product Lines 8
2.4.1 Use Case Model for Software Product Lines 9
2.4.2. Feature Analysis for Software Product Lines 9
2.4.3 Static Model for Software Product Lines 10
2.4.4 Collaboration Model for Software Product Lines 11
2.5 Other Software Product Line Engineering Methods 12
2.5.1 Feature-Oriented Domain Analysis (FODA) 12
2.5.2 Reuse-driven Software Engineering Method (RSEB) 13
2.5.3 FAST 13
2.5.4 KobrA 14
2.5.5 Knowledge-Based Requirement Elicitation Tool (KBRET) 14
2.5.6 Web-Based Software Product Lines 15
2.6 Component-Based Software Engineering 16
2.7 Web Services 19
2.7.1 Advantages of Web Services 20
2.7.2 Disadvantages of Web Services 21
2.7.3 Service-Oriented Architecture 21
2.8 Aspect-Oriented Programming 22
2.9 Frame Technology 24
2.10 Summary 26

3. PROBLEM STATEMENT AND RESEARCH APPROACH............27

3.1 Introduction........27
3.2 Problem Statement........28
3.3 Research Approach........28
3.4 Design Method for Software Product Line Service-Oriented Architecture........30
3.5 Development Environments........31
3.6 Proof-of-oncept Development Environment........34
3.7 Validation........37
3.8 Comparison with Other Approaches........38
3.8.1 Comparison with Other Software Architectures and Product Line Research..38
3.8.2 Comparison with Development Approaches and Tools........42
3.9 Summary........45

4. A DESIGN METHOD FOR SOFTWARE PRODUCT LINES BASED ON WEB SERVICES........46

4.1 Introduction........46
4.2 Design Architecture of SPL Engineering Phase........48
4.2.1 Use Case Modeling........49
4.2.2 Feature Modeling........52
4.2.3 User Interface Navigation Modeling........53
4.2.4 Interaction Modeling........56
4.2.5 Activity Modeling........58
4.2.6 Software Architecture Modeling........61
4.2.7 Attributes of Entity Classes........64
4.2.8 Design of Component Interfaces........64
4.3 Summary........67

5. DEVELOPMENT APPROACHES FOR PRODUCT LINE CUSTOMIZATION AND SEPARATION OF CONCERNS................68

5.1 Introduction........68
5.2 Dynamic Customization of Client Application........70
5.2.1 Development of DCAC Pattern........82
5.2.2 Advantages of DCAC Approach:........91
5.2.3 Disadvantages of DCAC Approach:........92
5.3 Introduction to the Customization Approaches Based on Separation of Concerns. 93
5.4 Development of Dynamic Customization of Client Application with Separation of Concerns........95
5.4.1 Development of DCAC-SC Pattern........103
5.4.2 Advantages and Disadvantages of DCAC-SC Approach:........111
5.5 Development of Static Customization of Client Application (SCAC) with Separation of Concerns........112
5.5.1 Development of SCAC Pattern........123
5.5.2 Advantages of SCAC Approach:........133

5.5.3 Disadvantages of SCAC Approach: 133
5.6 Comparison of Customization Methods 134
5.7 Usage of Development Approaches 135
5.8 Summary 136

6. SOFTWARE PRODUCT LINE ENVIRONMENT PROTOTYPE. 137

6.1 Introduction 137
6.2 Software Product Line Environment Prototype (SPLET) 138
6.2.1 Feature Modeling Subsystem: 143
6.2.2 Customization Subsystem: 156
6.2.3 Separation of Concerns and Source Code Integration Subsystem 171
6.2.4 Utility Subsystem 191
6.3 Validation 193
6.3.1 Validation Process 193
6.4 Summary 196

7. CONTRIBUTIONS AND FUTURE RESEARCH 197

7.1 Introduction 197
7.2 Research Contribution 197
7.3 Future Research 202
7.3.1 Testing of Software Product Lines 202
7.3.2 Transaction of Web Services Using Customizable Workflows 202
7.3.3 Performance of SPL Applications Based on Web Services 202
7.3.4 Evolution of SPL Applications Based on Web Services 203
7.4 Summary 203

APPENDIX A: RADIO FREQUENCY MANAGEMENT SYSTEM: A CASE STUDY 211

A.1 Introduction 211
A.2 Validation of This Research 212
A.3 Multiple-View Design Architecture 212
A.3.1 Use Case Modeling 213
A.3.2 Feature Modeling 216
A.3.3 User Interface Interaction Modeling 219
A.3.4 Detailed Design 221
A.3.5 Web Services Modeling 236
A.4. SPL Development 237
A.4.1 Dynamic Customization of Client Application (DCAC) Approach 238
A.4.2 Dynamic Customization of Client Application with Separation of Concerns (DCAC-SC) Approach 242
A.4.3 Static Customization of Client Application (SCAC) Approach 248
A.4.4 Summary 252

APPENDIX B: DEVELOPMENT ENVIRONMENT PATTERNS 253

B.1 Introduction 253
B.2 Dynamic Cient Application Customization Pattern 254
B.3 Dynamic Client Application Customization with Separation of Concerns Pattern 261
B.4 Static Client Application Customization Pattern 268

LIST OF FIGURES

Page
Figure 2-1 Evolutionary Software Product Line Engineering Process 8
Figure 2-2 Component-Based Design Pattern [Bachmann00] 17
Figure 2-3 Service-Oriented Architecture [Irek03] 22
Figure 2-4 Aspect-Oriented Programming Architecture [Anastasopoulos01] 23
Figure 2-5 Example of an x-frame hierarchy [Zhang03b] 25
Figure 3-1 SPLET components 36
Figure 4-1 Evolutionary Software Product Line Engineering Process 46
Figure 4-2 Use Case Diagram 51
Figure 4-3 Feature Dependency Model 53
Figure 4-4 User Interface Navigation Model 55
Figure 4-5 GUI –RoomReservation UI 56
Figure 4-6 Collaboration Diagram – Reserve single room 57
Figure 4-7 Expanded Collaboration Diagram – Reserve single room 58
Figure 4-8 Activity Diagram– Main Reservation 59
Figure 4-9 Activity Diagram – Overall Room Reservation UI 60
Figure 4-10 Activity Diagram–Reserve Room 61
Figure 4-11 Example of Web Services Grouping 62
Figure 4-12 Sample Input/Output for ReserveRoomWS 63
Figure 4-13 Sample Entity Attributes for ReserveRoomWS 64
Figure 4-14 Example of ports and connectors - RoomReservation Feature 65
Figure 4-15 Example of Ports, Provided, and Required Interfaces 66
Figure 4-16 Example of Port Interfaces Design 67
Figure 5-1 Conceptual Overview of DCAC Approach 71
Figure 5-2 Dynamic Customization Workflows (DCAC) Pattern 81
Figure 5-3 Activity Diagram - Main Reservation UI 82
Figure 5-4 Customization phase - Main Reservation UI 84
Figure 5-5 Activity Diagram – RoomReservation UI 87
Figure 5-6 Collaboration Diagram – RoomReservation 89
Figure 5-7 Implementation - RoomReservation UI 89
Figure 5-8 Conceptual Overview of DCAC-SC Approach 97
Figure 5-9 Dynamic Client Application Customization with Separation of Concerns Pattern 103
Figure 5-10 Activity Diagram - Main Reservation UI 104
Figure 5-11 MainReservation - Graphical User Interface 105
Figure 5-12 Implementation - Main Reservation UI 106

Figure 5-13 Implementation - Main Reservation UI 108
Figure 5-14 MainReservation UI - Insertion Points List 110
Figure 5-15 Conceptual overview of SCAC approach 113
Figure 5-16 Static Client Application Customization (SCAC) Pattern 122
Figure 5-17 Activity Diagram - Main Reservation UI 123
Figure 5-18 MainReservation - Graphical User Interface 124
Figure 5-19 Implementation - Main Reservation UI 126
Figure 5-20 Implementation - Main Reservation UI 129
Figure 5-21 Implementation - Main Reservation UI with RoomReservation Feature 130
Figure 5-22 Implementation - Main Reservation UI with ResidentialReservation Feature 131
Figure 6-1 Evolutionary Software Product Line Engineering Process 137
Figure 6-2 SPLET Components 141
Figure 6-3 Detailed Description of SPLET 142
Figure 6-4 SPLET - Main Screen 143
Figure 6-5 Feature Modeling Subsystem 144
Figure 6-6 Entity Class Diagram 145
Figure 6-7 Feature Editor-Main Interface 147
Figure 6-8 Feature Editor – Feature Creation 148
Figure 6-9 Feature Creation 148
Figure 6-10 Feature Dependency Tree 150
Figure 6-11 Feature Editor – Related Diagrams 150
Figure 6-12 Storing Related SPL Artifacts 151
Figure 6-13 Feature Editor – Parameterized Variables 152
Figure 6-14 Creation of Parameterized Variables 152
Figure 6-15 Feature Editor - Web Services 153
Figure 6-16 Adding Web Services 153
Figure 6-17 Web Service Editor 155
Figure 6-18 Adding Web Services 155
Figure 6-19 Customization Subsystem in SPLET 157
Figure 6-20 Feature Selector - Main Interface 158
Figure 6-21 Feature Selector - Customization 159
Figure 6-22 Feature Selector – Diagrams 161
Figure 6-23 Display Artifacts 161
Figure 6-24 Feature Selector - Related Web Services 162
Figure 6-25 Web Service Invocation 163
Figure 6-26 Web Service invocation - ReserveRoom 164
Figure 6-27 SOAP Message 165
Figure 6-28 Results Returned from roomReservation WS 166
Figure 6-29 Customization File - Generator Component 169
Figure 6-30 Entity Class Diagram - Customization File 170
Figure 6-31 Customization File Generation 170
Figure 6-32 SPLET – Separation of Concerns & Code Weaving 172
Figure 6-33 Variable Source Code Editor - Single Features 174

Figure 6-34 Single Feature Variable Source Code File Creation 174
Figure 6-35 Variable Source Code Editor - Multi Features 176
Figure 6-36 Multi Feature Variable Source Code File Editor 177
Figure 6-37 Variable Source Code Editor - Composed Features 178
Figure 6-38 Creation of Variable Source Code File 179
Figure 6-39 Code Tracker 181
Figure 6-40 Tracking of Feature Related Insertion Points 182
Figure 6-41 Tracking of Specific Insertion Point Name 183
Figure 6-42 Code Weaver 185
Figure 6-43 Dynamic Integration 186
Figure 6-44 Code Weaving for DCAC-SC Method 187
Figure 6-45 Static Integration 189
Figure 6-46 Code Weaving for SCAC Method 189
Figure 6-47 Samples of Variable File 191
Figure 6-48 SPLET - Utility 192
Figure 6-49 File Extractor Utility 192
Figure A-1 Use Case Model 214
Figure A-2 SPL Feature Model 218
Figure A-3 User Interface Interaction Model 220
Figure A-4 User Interface - MainUI 221
Figure A-5 Activity Diagram - MainUI User Interface 222
Figure A-6 Customization Phase - MainUI User Interface 223
Figure A-7 Interaction Modeling - MainUI user Interface 224
Figure A-8 Equipement/Antenna setup for MMC 225
Figure A-9 Activity Diagram – MMCconnect UI 226
Figure A-10 Collaboration Diagram - MMCconnect UI 226
Figure A-11 InterconnectionWS 227
Figure A-12 Equipment/Antenna Setup - RMS 228
Figure A-13 Activity Diagram – RMSconnect UI 229
Figure A-14 Collaboration Diagram - RMSconnect UI 229
Figure A-15 Equipment/Antenna Setup - MMS 230
Figure A-16 Activity Diagram – MMSconnect UI 231
Figure A-17 Collaboration Diagram - MMS 231
Figure A-18 Interference Measurement UI 232
Figure A-19 Activity Diagram – InterferenceMeasurement UI 233
Figure A-69 Customization Phase – InterferenceMeasurement UI 234
Figure A-21 Collaboration Diagram – Frequency Deviation 235
Figure A-22 Web Service Modeling 236
Figure A-23 Activity Diagram - MainUI User Interface 237
Figure A-24 DCAC Implementation - MainUI User Interface 238
Figure A-25 DCAC-SC Implementation - Main Reservation UI 243
Figure A-26 Integrated Source Code - MainUI 245
Figure A-27 SCAC Implementation - Main Reservation UI 249
Figure A-28 Integrated Source Code - MainUI 250

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

Software Product Line Engineering Based on Web Services

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctoral of Philosophy at George Mason University.

By

Mazen M. Aquil Saleh

Bachelor of Science, Texas Southern University, 1990
Master of Science, American University, 2000

Director: Dr. Hassan Gomaa
Professor, Information and Software Systems Engineering

Spring Semester 2005
George Mason University
Fairfax, Virginia

ABSTRACT

SOFTWARE PRODUCT LINE ENGINEERING BASED ON WEB SERVICES

Mazen Saleh, Ph.D.

George Mason University, 2005

Dissertation Director: Dr. Hassan Gomaa

The field of software reuse has evolved from reuse of individual components towards large-scale reuse with software product lines. A software product line (SPL) consists of a family of software systems that have some common functionality and some variable functionality. A family of systems is frequently referred to as a software product line or software product family.

This thesis investigates the technology of web services in the development and customization of software product lines. Web services are defined as a collection of software components that use XML to communicate with other applications over the Internet.

Based on a survey of SPL engineering methods and environments, current approaches do not address the design, development, and automatic customization of software product

lines based on web services. It is necessary to extend the current approaches for modeling single web services-based systems to address the unique issues of software product lines.

It is also necessary to introduce an automated development environment that enables developers to develop and automatically customize the web services-based software product line to generate executable target systems.

In order to solve this problem, this research develops a design approach for developing software product lines based on web services. The design approach is based on a multiple-view model for SPL. It addresses the unique issues of engineering a web service-oriented customizable software product line system.

This research also describes three different development approaches to develop the proposed SPL design for automatic customization. The first approach describes the development of a SPL application that can be customized dynamically at run time. The second approach is an extension to the first approach to include separation of concerns between variable source code and kernel source code. The third development approach describes the development of a SPL application that can be customized at source code integration time.

A proof-of-concept software product line engineering environment is developed to support the different development and customization approaches. The SPL engineering

environment supports the creation of a SPL model, customization of SPL applications based on each of the development approaches, and establishing separation of concerns and integration between variable source code and kernel source code.

SOFTWARE PRODUCT LINE ENGINEERING BASED ON WEB SERVICES

by

Mazen M. Aquil Saleh
A Dissertation Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctoral of Philosophy
Information Technology

Committee:

_____ Hassan Gomaa, Dissertation Director

_____ Larry Kerschberg

_____ Alex Brodsky

_____ Curtis Jamison

_____ Stephen G. Nash, Associate Dean for Graduate Studies and Research

_____ Lloyd J. Griffiths, Dean, School of Information Technology and Engineering

Date: 4/26/2005

Spring, 2005
George Mason University
Fairfax, Virginia

Dedication

I would like to dedicate this dissertation to my father, Mohammad, and mother, Ehsan, for all their support, encouragement, moral teachings, and sacrifices throughout my entire life and my PhD journey. My father has always been my role model for his wisdom and way of life. My mother has given me eternal love and care.

I also dedicate this dissertation to my beautiful wife, Sahar, and my children, Faris, Amani, and Reem. My wife has stood by my side since my undergraduate study. She has been my close friend and my life companion.

Acknowledgment

I would like to express my deep appreciation to my PhD. Advisor, Prof. Hassan Gomaa for his great guidance, encouragement, and support throughout my PhD program. He taught me all of what I know in the Software Product Line Engineering field. He made it possible for me to relate the academic teachings to real life working experience. He has always found time to work with me and keep me focus in my research. His guidance and critiques have influenced my research to produce quality work. I truly believe that he is the best advisor a student can have.

I would like to thank Dr. Kerschberg for his teachings and support. His method of teaching was outstanding. He opened for me many doors to new knowledge. His teachings had great influence in my research.

I would like to thank my entire dissertation committee for their encouragement and support.

Finally, I would like to thank Erika Olimpiew for her help and support. We have been in this PhD journey together from the beginning. Erika and I have started a PhD informal sessions that were held weekly with many graduate students to exchange information and obtain feedback. Her comments and critiques were invaluable.

TABLE OF CONTENTS

Page

ABSTRACT XI

1. INTRODUCTION 1

1.1 Background 1
1.2 Research Problem and Approach 2
1.3 Importance and Rationale of This Research 3
1.4 Terminology 3
1.5 Organization 5

2. RELATED WORK 6

2.1 Introduction 6
2.2 Software Product Lines 6
2.3 Evolutionary Software Product Line Engineering Process 7
2.4 Multiple-View Models of Software Product Lines 8
2.4.1 Use Case Model for Software Product Lines 9
2.4.2. Feature Analysis for Software Product Lines 9
2.4.3 Static Model for Software Product Lines 10
2.4.4 Collaboration Model for Software Product Lines 11
2.5 Other Software Product Line Engineering Methods 12
2.5.1 Feature-Oriented Domain Analysis (FODA) 12
2.5.2 Reuse-driven Software Engineering Method (RSEB) 13
2.5.3 FAST 13
2.5.4 KobrA 14
2.5.5 Knowledge-Based Requirement Elicitation Tool (KBRET) 14
2.5.6 Web-Based Software Product Lines 15
2.6 Component-Based Software Engineering 16
2.7 Web Services 19
2.7.1 Advantages of Web Services 20
2.7.2 Disadvantages of Web Services 21
2.7.3 Service-Oriented Architecture 21
2.8 Aspect-Oriented Programming 22
2.9 Frame Technology 24
2.10 Summary 26

3. PROBLEM STATEMENT AND RESEARCH APPROACH............27

3.1 Introduction....27
3.2 Problem Statement....28
3.3 Research Approach....28
3.4 Design Method for Software Product Line Service-Oriented Architecture....30
3.5 Development Environments....31
3.6 Proof-of-oncept Development Environment....34
3.7 Validation....37
3.8 Comparison with Other Approaches....38
3.8.1 Comparison with Other Software Architectures and Product Line Research..38
3.8.2 Comparison with Development Approaches and Tools....42
3.9 Summary....45

4. A DESIGN METHOD FOR SOFTWARE PRODUCT LINES BASED ON WEB SERVICES....46

4.1 Introduction....46
4.2 Design Architecture of SPL Engineering Phase....48
4.2.1 Use Case Modeling....49
4.2.2 Feature Modeling....52
4.2.3 User Interface Navigation Modeling....53
4.2.4 Interaction Modeling....56
4.2.5 Activity Modeling....58
4.2.6 Software Architecture Modeling....61
4.2.7 Attributes of Entity Classes....64
4.2.8 Design of Component Interfaces....64
4.3 Summary....67

5. DEVELOPMENT APPROACHES FOR PRODUCT LINE CUSTOMIZATION AND SEPARATION OF CONCERNS................68

5.1 Introduction....68
5.2 Dynamic Customization of Client Application....70
5.2.1 Development of DCAC Pattern....82
5.2.2 Advantages of DCAC Approach:....91
5.2.3 Disadvantages of DCAC Approach:....92
5.3 Introduction to the Customization Approaches Based on Separation of Concerns. 93
5.4 Development of Dynamic Customization of Client Application with Separation of Concerns....95
5.4.1 Development of DCAC-SC Pattern....103
5.4.2 Advantages and Disadvantages of DCAC-SC Approach:....111
5.5 Development of Static Customization of Client Application (SCAC) with Separation of Concerns....112
5.5.1 Development of SCAC Pattern....123
5.5.2 Advantages of SCAC Approach:....133

5.5.3 Disadvantages of SCAC Approach: 133
5.6 Comparison of Customization Methods 134
5.7 Usage of Development Approaches 135
5.8 Summary 136

6. SOFTWARE PRODUCT LINE ENVIRONMENT PROTOTYPE. 137

6.1 Introduction 137
6.2 Software Product Line Environment Prototype (SPLET) 138
6.2.1 Feature Modeling Subsystem: 143
6.2.2 Customization Subsystem: 156
6.2.3 Separation of Concerns and Source Code Integration Subsystem 171
6.2.4 Utility Subsystem 191
6.3 Validation 193
6.3.1 Validation Process 193
6.4 Summary 196

7. CONTRIBUTIONS AND FUTURE RESEARCH 197

7.1 Introduction 197
7.2 Research Contribution 197
7.3 Future Research 202
7.3.1 Testing of Software Product Lines 202
7.3.2 Transaction of Web Services Using Customizable Workflows 202
7.3.3 Performance of SPL Applications Based on Web Services 202
7.3.4 Evolution of SPL Applications Based on Web Services 203
7.4 Summary 203

APPENDIX A: RADIO FREQUENCY MANAGEMENT SYSTEM: A CASE STUDY 211

A.1 Introduction 211
A.2 Validation of This Research 212
A.3 Multiple-View Design Architecture 212
A.3.1 Use Case Modeling 213
A.3.2 Feature Modeling 216
A.3.3 User Interface Interaction Modeling 219
A.3.4 Detailed Design 221
A.3.5 Web Services Modeling 236
A.4. SPL Development 237
A.4.1 Dynamic Customization of Client Application (DCAC) Approach 238
A.4.2 Dynamic Customization of Client Application with Separation of Concerns (DCAC-SC) Approach 242
A.4.3 Static Customization of Client Application (SCAC) Approach 248
A.4.4 Summary 252

APPENDIX B: DEVELOPMENT ENVIRONMENT PATTERNS 253

B.1 Introduction 253
B.2 Dynamic Cient Application Customization Pattern 254
B.3 Dynamic Client Application Customization with Separation of Concerns Pattern 261
B.4 Static Client Application Customization Pattern 268

LIST OF FIGURES

Page
Figure 2-1 Evolutionary Software Product Line Engineering Process 8
Figure 2-2 Component-Based Design Pattern [Bachmann00] 17
Figure 2-3 Service-Oriented Architecture [Irek03] 22
Figure 2-4 Aspect-Oriented Programming Architecture [Anastasopoulos01] 23
Figure 2-5 Example of an x-frame hierarchy [Zhang03b] 25
Figure 3-1 SPLET components 36
Figure 4-1 Evolutionary Software Product Line Engineering Process 46
Figure 4-2 Use Case Diagram 51
Figure 4-3 Feature Dependency Model 53
Figure 4-4 User Interface Navigation Model 55
Figure 4-5 GUI –RoomReservation UI 56
Figure 4-6 Collaboration Diagram – Reserve single room 57
Figure 4-7 Expanded Collaboration Diagram – Reserve single room 58
Figure 4-8 Activity Diagram– Main Reservation 59
Figure 4-9 Activity Diagram – Overall Room Reservation UI 60
Figure 4-10 Activity Diagram–Reserve Room 61
Figure 4-11 Example of Web Services Grouping 62
Figure 4-12 Sample Input/Output for ReserveRoomWS 63
Figure 4-13 Sample Entity Attributes for ReserveRoomWS 64
Figure 4-14 Example of ports and connectors - RoomReservation Feature 65
Figure 4-15 Example of Ports, Provided, and Required Interfaces 66
Figure 4-16 Example of Port Interfaces Design 67
Figure 5-1 Conceptual Overview of DCAC Approach 71
Figure 5-2 Dynamic Customization Workflows (DCAC) Pattern 81
Figure 5-3 Activity Diagram - Main Reservation UI 82
Figure 5-4 Customization phase - Main Reservation UI 84
Figure 5-5 Activity Diagram – RoomReservation UI 87
Figure 5-6 Collaboration Diagram – RoomReservation 89
Figure 5-7 Implementation - RoomReservation UI 89
Figure 5-8 Conceptual Overview of DCAC-SC Approach 97
Figure 5-9 Dynamic Client Application Customization with Separation of Concerns Pattern 103
Figure 5-10 Activity Diagram - Main Reservation UI 104
Figure 5-11 MainReservation - Graphical User Interface 105
Figure 5-12 Implementation - Main Reservation UI 106

Figure 5-13 Implementation - Main Reservation UI.. 108
Figure 5-14 MainReservation UI - Insertion Points List.. 110
Figure 5-15 Conceptual overview of SCAC approach.. 113
Figure 5-16 Static Client Application Customization (SCAC) Pattern.. 122
Figure 5-17 Activity Diagram - Main Reservation UI.. 123
Figure 5-18 MainReservation - Graphical User Interface.. 124
Figure 5-19 Implementation - Main Reservation UI.. 126
Figure 5-20 Implementation - Main Reservation UI.. 129
Figure 5-21 Implementation - Main Reservation UI with RoomReservation Feature.... 130
Figure 5-22 Implementation - Main Reservation UI with ResidentialReservation Feature 131
Figure 6-1 Evolutionary Software Product Line Engineering Process.. 137
Figure 6-2 SPLET Components.. 141
Figure 6-3 Detailed Description of SPLET.. 142
Figure 6-4 SPLET - Main Screen.. 143
Figure 6-5 Feature Modeling Subsystem.. 144
Figure 6-6 Entity Class Diagram.. 145
Figure 6-7 Feature Editor-Main Interface.. 147
Figure 6-8 Feature Editor – Feature Creation.. 148
Figure 6-9 Feature Creation.. 148
Figure 6-10 Feature Dependency Tree.. 150
Figure 6-11 Feature Editor – Related Diagrams.. 150
Figure 6-12 Storing Related SPL Artifacts.. 151
Figure 6-13 Feature Editor – Parameterized Variables.. 152
Figure 6-14 Creation of Parameterized Variables.. 152
Figure 6-15 Feature Editor - Web Services.. 153
Figure 6-16 Adding Web Services.. 153
Figure 6-17 Web Service Editor.. 155
Figure 6-18 Adding Web Services.. 155
Figure 6-19 Customization Subsystem in SPLET.. 157
Figure 6-20 Feature Selector - Main Interface.. 158
Figure 6-21 Feature Selector - Customization.. 159
Figure 6-22 Feature Selector – Diagrams.. 161
Figure 6-23 Display Artifacts.. 161
Figure 6-24 Feature Selector - Related Web Services.. 162
Figure 6-25 Web Service Invocation.. 163
Figure 6-26 Web Service invocation - ReserveRoom.. 164
Figure 6-27 SOAP Message.. 165
Figure 6-28 Results Returned from roomReservation WS.. 166
Figure 6-29 Customization File - Generator Component.. 169
Figure 6-30 Entity Class Diagram - Customization File.. 170
Figure 6-31 Customization File Generation.. 170
Figure 6-32 SPLET – Separation of Concerns & Code Weaving.. 172
Figure 6-33 Variable Source Code Editor - Single Features.. 174

Figure 6-34 Single Feature Variable Source Code File Creation 174
Figure 6-35 Variable Source Code Editor - Multi Features 176
Figure 6-36 Multi Feature Variable Source Code File Editor 177
Figure 6-37 Variable Source Code Editor - Composed Features 178
Figure 6-38 Creation of Variable Source Code File 179
Figure 6-39 Code Tracker 181
Figure 6-40 Tracking of Feature Related Insertion Points 182
Figure 6-41 Tracking of Specific Insertion Point Name 183
Figure 6-42 Code Weaver 185
Figure 6-43 Dynamic Integration 186
Figure 6-44 Code Weaving for DCAC-SC Method 187
Figure 6-45 Static Integration 189
Figure 6-46 Code Weaving for SCAC Method 189
Figure 6-47 Samples of Variable File 191
Figure 6-48 SPLET - Utility 192
Figure 6-49 File Extractor Utility 192
Figure A-1 Use Case Model 214
Figure A-2 SPL Feature Model 218
Figure A-3 User Interface Interaction Model 220
Figure A-4 User Interface - MainUI 221
Figure A-5 Activity Diagram - MainUI User Interface 222
Figure A-6 Customization Phase - MainUI User Interface 223
Figure A-7 Interaction Modeling - MainUI user Interface 224
Figure A-8 Equipement/Antenna setup for MMC 225
Figure A-9 Activity Diagram – MMCconnect UI 226
Figure A-10 Collaboration Diagram - MMCconnect UI 226
Figure A-11 InterconnectionWS 227
Figure A-12 Equipment/Antenna Setup - RMS 228
Figure A-13 Activity Diagram – RMSconnect UI 229
Figure A-14 Collaboration Diagram - RMSconnect UI 229
Figure A-15 Equipment/Antenna Setup - MMS 230
Figure A-16 Activity Diagram – MMSconnect UI 231
Figure A-17 Collaboration Diagram - MMS 231
Figure A-18 Interference Measurement UI 232
Figure A-19 Activity Diagram – InterferenceMeasurement UI 233
Figure A-69 Customization Phase – InterferenceMeasurement UI 234
Figure A-21 Collaboration Diagram – Frequency Deviation 235
Figure A-22 Web Service Modeling 236
Figure A-23 Activity Diagram - MainUI User Interface 237
Figure A-24 DCAC Implementation - MainUI User Interface 238
Figure A-25 DCAC-SC Implementation - Main Reservation UI 243
Figure A-26 Integrated Source Code - MainUI 245
Figure A-27 SCAC Implementation - Main Reservation UI 249
Figure A-28 Integrated Source Code - MainUI 250

1. INTRODUCTION

1.1 Background

The field of software reuse has evolved from reuse of individual components towards large-scale reuse with software product lines [Clements02]. A software product line (SPL) consists of a family of software systems that have some common functionality and some variable functionality. Parnas referred to a collection of systems that share common characteristics as a *family of systems* [Parnas79]. According to Parnas, it is worth considering the development of a family of systems when there is more to be gained by analyzing the systems collectively rather than separately, i.e. the systems have more features in common than features that distinguish them. A *family of systems* is now referred to as a *software product line* or *software product family.*

A Software Product Line (SPL) is developed by engineering a reusable architecture for the product line, which can be configured to generate target applications [Gomaa99, Gomaa04]. The two major activities used in developing product lines are SPL engineering and application engineering. SPL engineering involves the analysis, design, and implementation of product line software that satisfy the requirements of the families of systems [Weiss99, Gomaa04]. Application engineering involves tailoring the

engineered SPL to produce target applications based on a given set of configuration requirements [Sugumaran92, Gomaa04].

This dissertation addresses product lines based on web services. A web service is defined as a collection of functional methods that are grouped into a single package and published in the Internet for use by other applications. Web services use the standard Extensible Markup Language (XML) to exchange information with other software via the Internet protocols [Deitel et al. 2003, Howard04, Booth04].

Although there is much research into software product line engineering, this research extends product line concepts to address the engineering and customization of product lines that are based on web services.

1.2 Research Problem and Approach

This research focuses on designing, developing and customizing software product lines based on web services to derive executable target applications from the product line using an automated customization environment. The approach taken is to:

a) Develop a design approach for software product line service-oriented architecture.
b) Introduce three different development approaches to support the automatic customization of SPL architecture and components:
c) Develop a proof-of-concept prototype to support this research

d) Validate this research with two web services-based software product line case studies.

1.3 Importance and Rationale of This Research

The idea of web services has been strongly promoted in industry by companies such as Microsoft, IBM, Oracle, and Hewlett-Packard. They see this new technology as a broad new vision for how software systems are analyzed, developed, and used [McDougall 01]. Web services employ open standards that are text-based, which introduce a new approach to communication between heterogeneous platforms and applications [Deitel 03]. Using the already existing internet technology, web services make communication, interoperability, and integration cheaper and easier to achieve, compared to current methods, such as CORBA and DCOM [Deitel 03]. As the use of web services continues to grow, software product lines engineers should take full advantage of this technology. Therefore, it is essential to develop a new methodology that enables the design, development, and customization of software product lines that consist of web services-based components.

1.4 Terminology

This section provides definitions of important terms used in this dissertation.

Unified Modeling Language

Unified Modeling Language (UML) is a standardized object-oriented development environment that is used to analyze and design systems.

Software Product Line

A software product line (SPL) is a family of systems that share common features. It is developed by engineering an application domain that can be configured to generate target systems through the customization process of selecting optional and alternative features. [Parnas79, Gomaa04]

Feature

A feature is a functional requirement of a software application.

SPL Engineer

The SPL engineer is responsible for designing and developing the product line.

Application Engineer

The application engineer is responsible for customizing the product line to derive target applications.

Kernel Source Code

Kernel source code refers to source code that exists in all derived target applications.

Variable Source Code

Variable source code refers to optional or alternative source code blocks that are integrated with kernel source code based on feature selection to produce a customized target application.

Separation of Concerns

Separation of concerns refers to the separation of common and variable product line concerns. It involves the separation of variable source code from kernel source code into a variable source code file.

Code Weaving

Code weaving is the integration of kernel source code with optional and alternative source code

Client application

Client application refers to the client subsystem and the software objects it contains.

Server application

Server application refers to the server subsystem and its constituent web service components and database.

1.5 Organization

The rest of the dissertation is organized as follows. Chapter 2 contains an overview of related work. Chapter 3 addresses the problem statement and research approach, including comparison of related work with this research effort. Chapter 4 describes the proposed design approach using a Hotel System case study. Chapter 5 describes the three development approaches and their customization environment. Chapter 6 describes the proof-of-concept prototype that is used to support this research. Chapter 7 includes contributions and future research. References and appendices are attached at the end, including the second case study of Radio Frequency Management System.

2. RELATED WORK

2.1 Introduction

This chapter surveys other research efforts that are related to the research described in this dissertation. This chapter begins by defining software product lines in section 2.2. Section 2.3 describes the Evolutionary Software Product Line Engineering Process (PLUS). Section 2.4 describes the multiple-view model of software product lines used in the PLUS environment. Section 2.5 addresses other software product line engineering methods. 2.6 describes component-based software engineering. Web services are described in section 2.7. Section 2.8 describes Aspect-Oriented Programming, and section 2.9 describes frame technology.

2.2 Software Product Lines

A software product line is a family of systems that share common features [Gomaa92, Gomaa04]. It is developed by engineering a Software Product Line (SPL) that can be tailored to generate target systems [Gomaa99, Farrukh98, Weiss99]. Software product line engineering involves the analysis, design, and implementation of a product line that satisfies the requirements of all target applications [Sugumaran92, Gomaa04]. This can be achieved by capturing the commonality and variability of a family of system at the analysis phase, and applying this information at the design and implementation phases

[Gomaa 99]. "The goal of software product families is to improve productivity through software reuse. A new application system can be configured from the domain model given the common features (requirement) of the domain and variable features that reflect differences among the members of the product family" [Farrukh 1998].

2.3 Evolutionary Software Product Line Engineering Process

The Evolutionary Software Product Line Engineering Process (PLUS) [Gomaa04] consists of two main processes, as shown in Figure 2-1:

a) Software Product line Engineering. A product line multiple-view model, which addresses the multiple views of a software product line, is developed. The product line multiple-view model, product line architecture, and reusable components are developed and stored in the product line reuse library.

b) Application engineering. Involves the configuration of target applications from the SPL architecture and implementation. A target application is a member of the software product line. The multiple-view model for a target application is configured from the product line multiple-view model. The user selects the desired features for the product line member (referred to as target application). Given the target application features, the product line model and architecture are adapted and tailored to derive the target application model and architecture. The architecture determines which of the reusable components are needed for configuring the executable target application.

Earlier papers have described how this approach was carried out before [Gomaa96, Gomaa99] and after the introduction of the UML [Gomaa02, Gomaa04]. This research describes how product line engineering can be carried out for product lines that are based on Web Services.

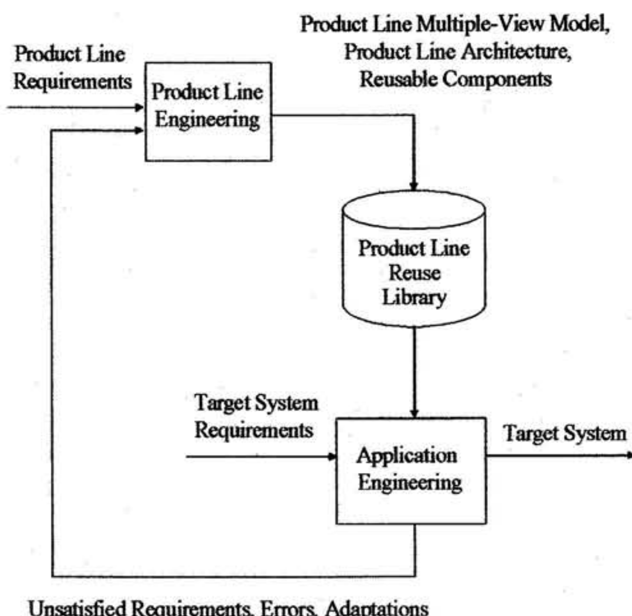


Figure 2-1 Evolutionary Software Product Line Engineering Process

2.4 Multiple-View Models of Software Product Lines

A multiple-view model for a software product line defines the different characteristics of a software family [Parnas79], including the commonality and variability among the members of the family [Clements02, Weiss99]. A multiple-view model is represented using the UML notation [Rumbaugh99, Gomaa00a, Gomaa04] and considers the product

line from different perspectives. The PLUS environment [Gomaa04] is based on the multiple-view mode for software product lines, as described in the following sections.

2.4.1 Use Case Model for Software Product Lines

The functional requirements of a system are defined in terms of use cases and actors [Jacobson97]. An actor is a user type. A use case describes the sequence of interactions between the actor and the system, considered as a black box.

For a single system, all use cases are required. When modeling a software product line, kernel use cases are those use cases required by all members of the family. Optional use cases are those use cases required by some but not all members of the family. Some use cases may be alternative, that is different versions of the use case are required by different members of the family [Gomaa04].

2.4.2. Feature Analysis for Software Product Lines

Feature analysis is an important aspect of domain analysis [Cohen98, Gomaa04, Griss98, Kang90]. In domain analysis, features are analyzed and categorized as kernel features (must be supported in all target systems), optional features (only required in some target systems), and prerequisite features (dependent upon other features). There may also be dependencies among features, such as mutually exclusive features. The emphasis in feature analysis is on the optional and alternative features, since they differentiate one member of the family from the others. In modeling software product lines, features may be functional features (addressing software functional requirements), non-functional

features (e.g., relating to security or performance), or parametric features (e.g., parameter whose value can be set differently in different members of the product line).

In the object-oriented analysis of single systems, use cases are used to determine the functional features of a system. They can also serve this purpose in product families. Griss [Griss98] has pointed out that the goal of the use case analysis is to get a good understanding of the functional requirements whereas the goal of feature analysis is to enable reuse. Use cases and features may be used to complement each other. In particular, use cases can be mapped to features based on their reuse properties.

Functional requirements that are required by all members of the family are packaged into a kernel feature. From a use case perspective, this means that the kernel use cases, which are required by all members of the family, constitute the kernel feature. Optional use cases, which are always used together, may also be packaged into an optional feature.

2.4.3 Static Model for Software Product Lines

A static model for a product line has kernel classes, which are used by all members of the product family, and optional classes that are used by some but not all members of the family. Variants of a class, which are used by different members of the product family, can be modeled using a generalization / specialization hierarchy. UML stereotypes are used to allow new modeling elements, tailored to the modeler's problem, which are based on existing modeling elements [Booch99, Rumbaugh99]. Thus, the stereotypes

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

References

[Anastasopoulos01] M. Anastasopoulos and C. Gacek. "Implementing Product Line Variabilities," Proc ACM Symposium on Software Reusability, Toronto, May 2001, pp. 109-117.

[Anastasopoulos04] M. Anastasopoulos and D. Muthig, "An Evolution of Aspect-Oriented Programming as a Product Line Implementation Technology," Proc. 8th International Conference on Software Reuse, Springer LNCS 3107, 2004, pp. 141- 156.

[Atkinson00] C. Atkinson, J. Bayer, and D. Muthig, "Component-Based Product Line Development: The Kobra Approach," SPCL, 2000. Available: http://se2c.uni.lu/tiki/se2c-bib_download.php?id=700.

[Bachmann00] F. Bachmann, L., C. Buhman, S. Comella-Dorda et al., "Technical Concepts of Component-Based Software Engineering. 2000," Software Engineering Institute, Carnegi Mellon University, Pittsburgh, PA, May 2000.

[Bass00] L. Bass, C. Buhman, S. Comella-Dorda et al., "Market Assesment of Component-Based Software Engineering," Software Engineering Institute, Carnegi Mellon University, 2000.

[Baxter01] I. Baxter, "Dms(the Design Maintenance System) a Tool for Automating Software Quality Enhancement," Semantic Designs, Inc, 2001.

[Bassett97] P. Bassett, Framing Software Reuse – Lessons from the Real World, Prentice Hall, 1997.

[Bisson] S. Bisson, "At Your Service", DNJ, 2004. Available: http://www.dnjonline.com/articles/architect/may04_atyourservice.asp.

[Bodkin02] R. Bodkin, "Commercialization AOSD: The Road Ahead," 2002. Available: http://www.jpmdesign.de/conferences/aosd/2003/papers/AOSD_Commercialization_Position_2003_final.pdf.

[Boonsiri02] S. Boonsiri, "Automated Component Ensamble Evaluation," International Journal of Information technology, Vol 8, No. 1, 2002.

[Booth04] D. Booth, et al. "Web Services Architecture," W3C, 2004. Available http://www.w3.org/TR/2003/WD-ws-arch-20030514.

[Chung03] J. Chung, K. Lin, and R. Mathieu. "Web Services Computing: Advancing Software Interoperability," IEEE Computer Society, Vol. 36, 2003, pp. 35-37.

[Clements02] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns, Addison Wesley, 2002.

[Cohen98] S. Cohen and L. Northrop, "Object-Oriented Technology and Domain Analysis," Proc. International Conference on Software Reuse, Victoria, June 1998.

[Coplien98] J. Coplien, D. Hoffman, and D. Weiss. "Commonality and Variability in Software Engineering," IEEE Software, 1998, Vol 15, No. 6, pp. 37-45.

[Deitel03] H. Deitel, B. DuWaldt, et al. Web Services - A technical Introduction. Upper Saddle River, New Jersey, Pearson Education, Inc, 2003.

[Fontana01] J. Fontana, "Microsoft, Sun Propel Web Services," Network World, Vol18, No.1, 2001, pp. 8-10.

[Farrukh98] G. Farrukh, "A Method and Software Engineering Environment for Configuring Applications from Reusable Specifications and Architectures," PhD Dissertation. George Mason University, 1998.

[Friedlande02] P. Friedlander, D. Collins, "Component-Based Software Development and the Software Factory," Infotech Update, Vol10, No2, 2002, pp. 4-9.

[Gladwin01] L. Gladwin, "Web Driving Demand for Integrated Apps," Computerworld, Vol. 35, No. 17, 2001, p. 56.

[Gomaa96] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I Tavakoli, "A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures," J. Automated Software Eng, Vol. 3, Nos. 3/4, 1996.

[Gomaa96a] H. Gomaa, D. Menasce, and L.Kerschberg. "A Software Architectural Design Method for Large-Scale Distributed Information Systems," Distrib. Syst. Eng, 1996, Vol. 3, No. 3, pp. 162-172.

[Gomaa99] H. Gomaa and G.A. Farrukh, "Methods and Tools for the Automated Configuration of Distributed Applications from Reusable Software Architectures and Components," IEE Proceedings – Software, Vol. 146, No. 6, December 1999.

[Gomaa00] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison Wesley, Reading MA, 2000.

[Gomaa00a] H. Gomaa, L. Kerschberg, G. Farrukh. "Domain Modeling of Software Process Model," IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, Tokyo, Japan, September 2000.

[Gomaa02] H. Gomaa and Michael E. Shin, "Multiple-View Meta-Modeling of Software Product Lines," the Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Maryland, December, 2002.

[Gomaa04] H. Gomaa, Designing Software Product Lines: From Use Cases to Pattern-based Software Architectures with UML 2.0, Addison-Wesley, July 2004.

[Govatos02] G. Govatos, "UDDI is Yellow Pages of Web Services," Network World, Vol. 19, No. 21, pp. 41, 2002.

[Greenwood04] P. Greenwood, N. Loughran, L. Blair, A. Rashid, "Dynamic Framed Aspects for Dynamic Software Evolution," 2004. Available: http://www.comp.lancs.ac.uk/computing/aose/papers/dynFr_ramse04.pdf.

[Griss98] M. Griss, J. Favaro, M. D'Alessandro, "Integrating Feature Modeling with the RSEB," Proc. International Conference on Software Reuse, Victoria, June 1998.

[Hasimi03] S. Hashimi, "Service-Oriented Architecture Explained," 2003. Available: http://www.ondotnet.com/pub/a/dotnet/2003/08/18/soa_explained.html.

[Hao03] H. He, "What Is Service-Oriented Architecture," 2003. Available: http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html.

[Holmes03] C. Holmes, A. Evans, " A Review of frame Technology," 2003. Available: http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-369.pdf.

[Howard04] R. Howard, L. Kerschberg, "A Framework for Dynamic Semantic Web Services Management," Int. J. Cooperative Inf. Syst., Vol. 13, No. 4, 2004, pp. 441-85.

[Hussein03] M. Hussein, "A Software Architecturebased Method and Framework for the Design of Dynamically Reconfigurable Product Line Software Architectures," PhD Dissertation. George Mason University, 2003.

[Irek03] C. Irek, "Realizing a Service-Oriented Architecture With .Net," 2003. Available: http://www.15seconds.com/issue/031215.htm.

[Jacobson97] I. Jacobson, M. Griss, P. Jonsson, Software Reuse - Architecture, Process and Organization for Business Success, Addison Wesley, 1997.

[Jarzabek03] S. Jarzabek, P. Bassett, H. Zhang, W. Zhang", "XVCL: XML-based Variant Configuration Language," ICSE 2003, pp. 810-811.

[Kang 90] K. Kang et. al., "Feature-Oriented Domain Analysis," Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[Kirtland99] M. Kirtland, "Designing Component-Based Applications," Redmond, Washington, Microsoft Press, 1999.

[Lawson 03] A. Lawson, "Semantic Designs - Design Maintenance System Software Reengineering Toolkit," Semantic Designs, Inc. 2003. Available: http://www.semdesigns.com/Company/Publications/Semantic%20Designs%20-%20DMS%20SRT%201-1%20(TA000243APM).pdf.

[Lee02] K. Lee, W. Kuen. "An Introduction to Aspect-Oriented Programming," COMP 610E, 2002.

[Lesaint04] D. Lesaint and G. Papamargarittis, "Aspects for Synthesizing Applications by Refinement," LNCS 3107, 2004, pp. 115-126.

[Lesiecki02] N. Lesiecki, "Improve Modularity with Aspect-Oriented Programming," IBM, 2002. Available: http://www-106.ibm.com/developerworks/library/j-aspectj/.

[Loughran04a] N. Loughran and A. Rashid, "Framed Aspects: Supporting Variability and Configurability for AOP", Proc. 8th International Conference on Software Reuse, Springer LNCS 3107, 2004, pp. 127-140.

[Loughran04b] N. Loughran, A. Rashid, W. Zhang, S. Jarzabek, "Supporting Product Line Evolution with Framed Aspects," 2004. Available: http://www.comp.lancs.ac.uk/computing/users/loughran/ACP4IS%5Bfinal%5D.pdf

[McDougall 01] P. McDougall, J. Levitt. "Decoding Web Services," InformationWeek, issue 857, 2001, pp. 28-37.

[Mills00] K. Mills, Hassan Gomaa. "A Knowledge-Based Method for Inferring Sematic Concepts from Visual Models of System Behavior," ACM Transaction, Vol. 9, No. 3, 2000, pp. 306-37.

[O'Hara98] O'Hara-Schettino, Elizabeth, and Hassan Gomaa. "Dynamic Navigation in Multiple View Software Specification and Design," The Journal of Systems and Engineering, Vol 41, 1998, pp. 93-103.

[Pappalarado 01] D. Pappalarado "Start-Ups Aim to Manage Web Services," Network World, Vol. 10, No. 40, 2001, pp. 1-2.

[Rojak 96] S. Rojak, "Domains in Logical Data Modeling," DBMS Online, 1996. Available: http://www.dbmsmag.com/9603d14.html.

[Seacord03] R. Seacord, K. Nwosu, "Life Cycle Activity Areas for Component-Based Software Engineering Process," Carnegie Mellon University and Lucent Technology, Inc. 2003. Available: http://www.sei.cmu.edu/cbs/tools99/lifecycle/index.html.

[Shaw96] M. Shaw, D. Garlan, Software Architecture: Perspective on an Emerging Discipline, Upper Saddle River, New Jersey, Printice-Hall, 1996.

[Shin02] E. Shin, "Evolution in Multiple-View Models of Software Product Families," PhD Dissertation, George Mason University, 2002.

[Sodhi99] J. Sodhi, P. Sodhi, Software Reuse: Domain Analysis and Design Process, McGraw-Hill, New York, 1999.

[Sugumaran92] V. Sugumaran, H. Gomaa, and L. Kerschberg, "Generating Target System Specifications from a Domain Model Using Clips," Clips Conference Proceedings, Houston TX, Vol. 1, 1992, pp. 209-26.

[Parnas79] Parnas D., "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, March 1979.

[Rumbaugh99] J. Rumbaugh, G. Booch, I. Jacobson, The Unified Modeling Language Reference Manual, Addison Wesley, Reading MA, 1999.

[Vizard 01] M. Vizard, "Reuse Grail Is in Sight with Web Services," InfoWorld, Vol. 23, No. 38, 2001, p. 8.

[Weiss99] D M Weiss and C T R Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison Wesley, 1999.

[Zhang 03] H. Zhang, S. Jarzabek, "An XVCL-Based Approach to Software Product Line Development," Int. Conf. on Software Engineering and Knowledge Engineering, 2003.

العنوان: Software Product Line Engineering Based on Web Services

المؤلف الرئيسي: Saleh, Mazen M. Aquil

مؤلفين آخرين: Gomaa, Hassan(Super.)

التاريخ الميلادي: 2005

موقع: فيرفاكس، فرجينيا

رقم MD: 618453

نوع المحتوى: رسائل جامعية

اللغة: English

الدرجة العلمية: رسالة دكتوراه

الجامعة: George Mason University

الكلية: Volgenau School of Engineering

الدولة: الولايات المتحدة الأمريكية

قواعد المعلومات: Dissertations

مواضيع: البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات

رابط: https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

Appendix A: Radio Frequency Management System: A Case Study

A.1 Introduction

The radio frequency spectrum is used for a very wide variety of wireless communications. It is considered a valuable resource that needs to be managed efficiently. The Radio Frequency Management system (**RFMS**) is built to manage the distribution of frequencies and to discover frequency interferences and illegal transmissions. The RFMS is a software product line that serves different types of monitoring stations: main monitoring center (MMC), regional monitoring stations (RMS), and mobile monitoring stations (MMS). The RFMS is the second case study for software product lines based on web services that is used to validate this research. In this case study, a RFMS product line is to be created for different types of monitoring stations, which can be customized to the needs of individual stations. The RFMS case study applies the software design approach and the three development environments that are introduced in this research to create the SPL application.

The Radio Frequency Management System includes licensing of radio frequencies, advance interference calculations, and monitoring radio frequency transmissions to ensure compliance with national assignments and regulations. The system comprises of a

main monitoring center and many regional monitoring stations scattered around the country. The software operates locally at each station with remote operation facilities from the MMC to all other stations. The software for local operations is also available in the mobile monitoring stations, which support the activities of the fixed MMC and RMS stations.

A.2 Validation of This Research

The Radio Frequency Management System case study validates:

a) Multiple-view design architecture for SPL applications based on web services.

b) The three development approaches:

   - Dynamic Client Application Customization (DCAC).
   - Dynamic Client Application Customization with Separation of Concerns (DCAC-SC).
   - Static Client Application Customization (SCAC).

c) The proof-of-concept development environment prototype SPLET.

A.3 Multiple-view Design Architecture

This section describes the software product line modeling approach for the RFMS product lines based on Web Services.

A.3.1 Use Case Modeling

Figure A-1 depicts the Use Case diagram for the RFMS SPL, which captures the overall software requirements. The Use Cases are categorized as kernel, optional, or alternative as given by the PLUS method [Gomaa04].

The actors for this use case model are the users of the product line, providing inputs to a product line member system and receiving outputs from it.

- **Monitoring technician** – Performs actions pertaining to frequency occupancy monitoring, remote monitoring, and occupancy evaluation.
- **Monitoring engineer** – Performs actions pertaining to frequency analysis, frequency allocation, and interference measurement.
- **Data entry clerk** – Performs actions pertaining to data entry of frequency allocation.

Briefly, the use cases are:

- **Radio Frequency Occupancy:** Monitoring technician can monitor the spectrum for radio frequency occupancy for a period of time. All transmissions within the given range are detected and stored for analysis.
- **Frequency Occupancy Evaluation:** The result of the frequency occupancy is compared to the frequency management database for illegal transmissions.

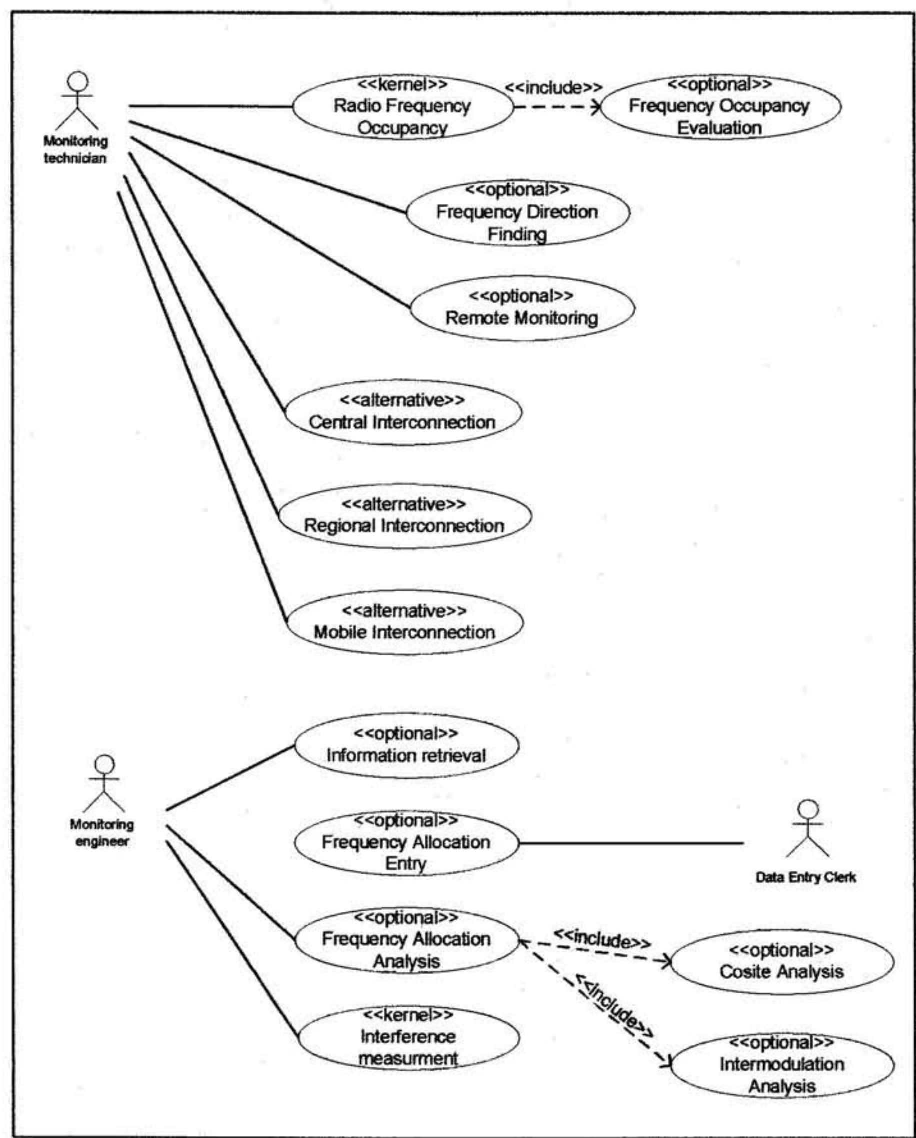


Figure A-1 Use case model

- **Remote Monitoring:** Monitoring technician in the main monitoring station uses the regional monitoring stations to perform remote frequency occupancy tasks.
- **Central Interconnection:** Monitoring technician can perform automatic setting of monitoring equipments and their related antennas for the main monitoring center.

- **Regional Interconnection:** Monitoring technician can perform automatic setting of monitoring equipments and their related antennas for the regional monitoring stations.
- **Mobile Interconnection:** Monitoring technician can perform automatic setting of monitoring equipments and their related antennas for the mobile monitoring stations.
- **Information Retrieval:** Monitoring engineer can retrieve technical information on allocated frequencies and retrieve administrative information regarding licensed users.
- **Frequency allocation entry:** Data entry clerk enters preliminary frequency data to be analyzed by the frequency engineer.
- **Frequency allocation analysis:** Monitoring engineer apply Electro Magnetic Compatibility (EMC) analysis on the proposed frequency request and take action whether to allocate the frequency, or reject it. The EMC analysis is based on arithmetic calculations of a proposed frequency against the already allocated frequencies in the frequency management database to avoid interference between assigned frequencies.
- **Cosite Analysis:** The Frequency allocation analysis may include interference analysis for frequencies that transmit from the same location.
- **Intermodulation Analysis:** The Frequency allocation analysis may include interference analysis for frequencies that may cause interference when they are modulated with other frequencies in the same coverage area.

- **Interference measurement:** The monitoring engineer process an interference complaint by coordinating with the monitoring technician to monitor the spectrum for all transmissions within a given range, and then perform interference measurement tests on suspicious frequencies. Interferences are usually caused by illegal transmissions, malfunction of transmitters causing frequency deviation, or signal level increase.
- **Frequency Direction finding:** Monitoring technician uses the mobile monitoring station to locate the source of a transmitter. Finding the direction of a transmission is part of the interference measurement tests.

A.3.2 Feature Modeling

A feature dependency model is derived from the use case model. Product line features are categorized as kernel, optional, or alternative features. Table A-1 shows the feature / use case dependencies based on the PLUS environment [Gomaa04].

Feature Name	Feature Category	Use Case Name	Use Case Category / Variation Point (VP)	Variation Point Name
MMC Interconnection	Alternative	Central Interconnection	Alternative	
RMS Interconnection	Alternative	Regional Interconnection	Alternative	
MMS Interconnection	Alternative	Mobile Interconnection	Alternative	

Table A-1 Feature / Use Case Dependencies

Feature Name	Feature Category	Use Case Name	Use Case Category / Variation Point (VP)	Variation Point Name
Frequency Occupancy	Kernel	Radio Frequency Occupancy	Kernel-VP	Equipment & Antenna types
Remote Occupancy	Optional	Remote Monitoring	Optional-VP	Equipment & Antenna types
Occupancy Evaluation	Optional	Frequency Occupancy Evaluation	Optional	
Interference MeasuremntCalc	Kernel	Interference Measurement	Kernel-VP	Equipment & Antenna types
Information Retrieval	Optional	Information Retrieval	Optional	
Frequency Allocation	Optional	Frequency Allocation Entry	Optional	
EMC Frequency Analysis	Optional	Frequency Allocation Analysis	Optional	
Co-Site Analysis	Optional	Cosite Analysis	Optional	
Inter-Modulation Analysis	Optional	Intermodulation Analysis	Optional	
Direction Finding	Optional	Frequency Direction Finding	Optional	

Table A-1 Feature / Use Case Dependencies (Continue)

The feature model in figure A-2 depicts the features of the SPL application.

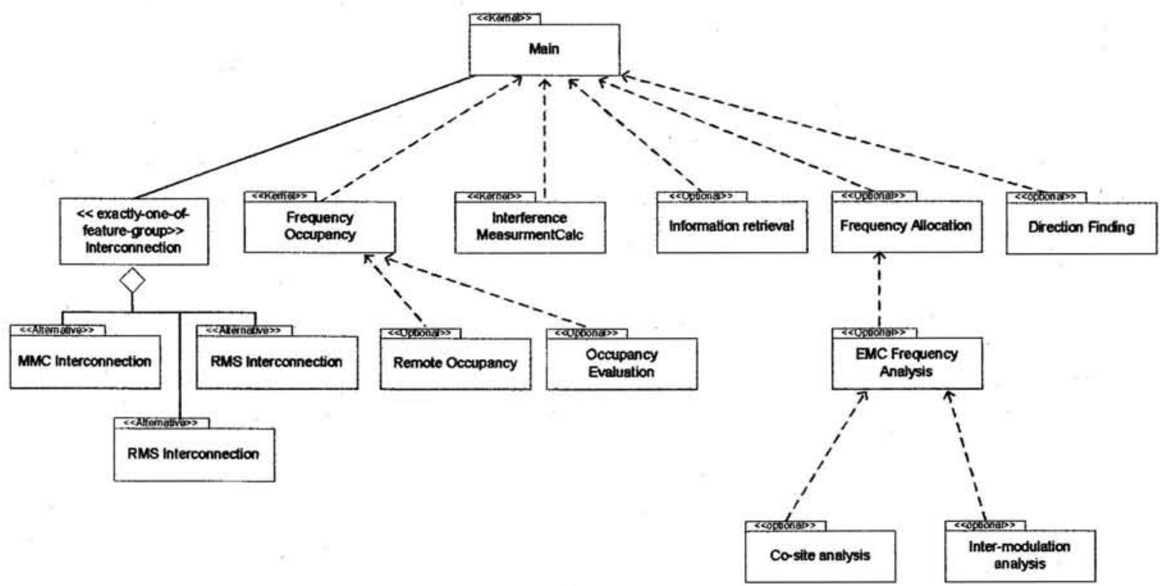


Figure A-2 SPL feature model

The SPL feature model is used as the main driver for customizing the SPL application. This model is entered in the SPLET tool to create the feature navigation tree. The feature model is used in SPLET to organize all SPL engineering components into their related features. The Feature Selector component in SPLET is used to select optional and alternative features from the feature tree when customizing target applications.

A.3.3 User interface Interaction Modeling

Since this design method is based on a service-oriented architecture for product lines, it is important to show the navigation between user interface screens. The navigation model is depicted from the feature model. Table A-2 shows the feature / class dependencies in the navigation model.

Feature Name	Feature Category	Class Name	Class Category	Class Parameter
Main	Kernel	MainUI Customizer	Kernel-VP Kernel	Title: String
MMC Interconnection	Alternative	MMCconnect	Alternative	
RMS Interconnection	Alternative	RMSconnect	Alternative	
MMS Interconnection	Alternative	MMSconnect	Alternative	
Frequency Occupancy	Kernel	FreqOccupancy Customizer	Kernel-VP Kernel	StationName: String
Remote Occupancy	Optional	RemoteOccupancy Customizer	Optional-VP Kernel	StationName: String
Occupancy Evaluation	Optional	OccupancyEvaluation	Optional	
Interference MeasuremntCalc	Kernel	InterferenceMeasurement Customizer	Optional Kernel	
Information Retrieval	Optional	InfoRetrieval FrequencyRetrieval UserRetrieval	Optional Optional Optional	
Frequency Allocation	Optional	FrequencyAlloc Acceptance	Optional Optional	
EMC Frequency Analysis	Optional	EMCAnalysis	Optional	

Table A-2 Feature / Class Dependencies

Feature Name	Feature Category	Class Name	Class Category	Class Parameter
Co-Site Analysis	Optional	EMCAnalysis	Optional	
Inter-Modulation Analysis	Optional	EMCAnalysis	Optional	
Direction Finding	Optional	DirectionFind Customizer	Optional-VP Kernel	StationName: String

Table A-2 Feature / Class Dependencies (Continue)

Figure A-3 is a user interface interaction model. It shows the navigation between user interfaces. Each user interface screen is supported by a user interface object, which is in turn associated with one or more Web services. Each user interface object contains a GUI and a customizable workflow for members of the software product line. The GUI will be responsible for accepting user input and user requests to initiate events that are translated into method calls to web services. After receiving the user input, the user interface object interacts with the appropriate Web service.

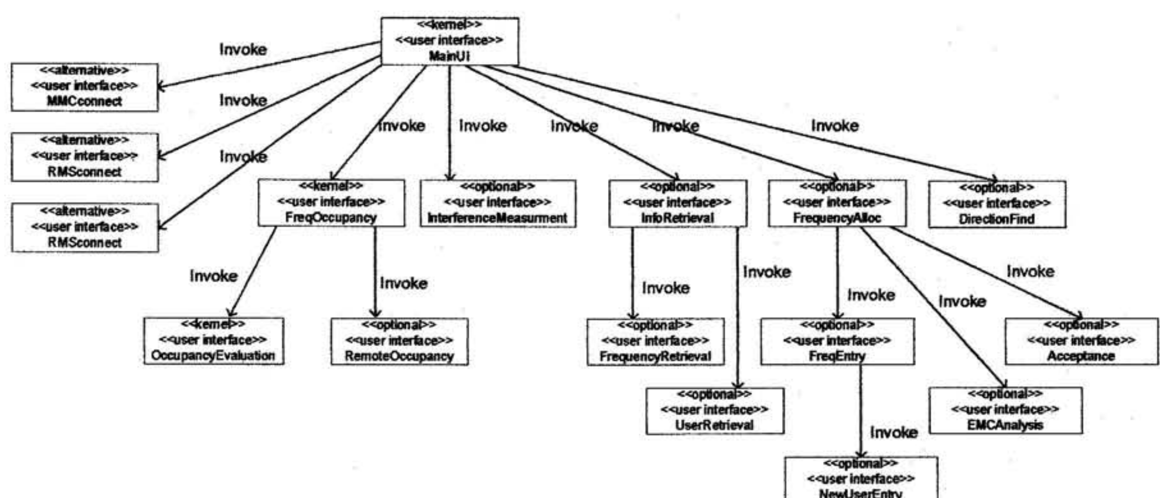


Figure A-3 User interface interaction model

A.3.4 Detailed Design

A.3.4.1 Main user interface

Figure A-4 shows a sample GUI for the "MainUI" user interface class. Figure A-5 shows a customizable activity diagram for the "MainUI" user interface. This diagram shows "MMCconnect", "RMSconnect", and "MMSconnect" user interfaces for equipment/antenna connection setup as mutually exclusive alternatives where only one of them can be invoked by clicking the Equipment Setup button of "MainUI" user interface (Figure A-4). "FreqOccupancy" and "InterferenceMeasurment" are kernel user interfaces. The diagram also shows the optional user interfaces: "InfoRetrieval", "FrequencyAlloc", and "DirectionFind".

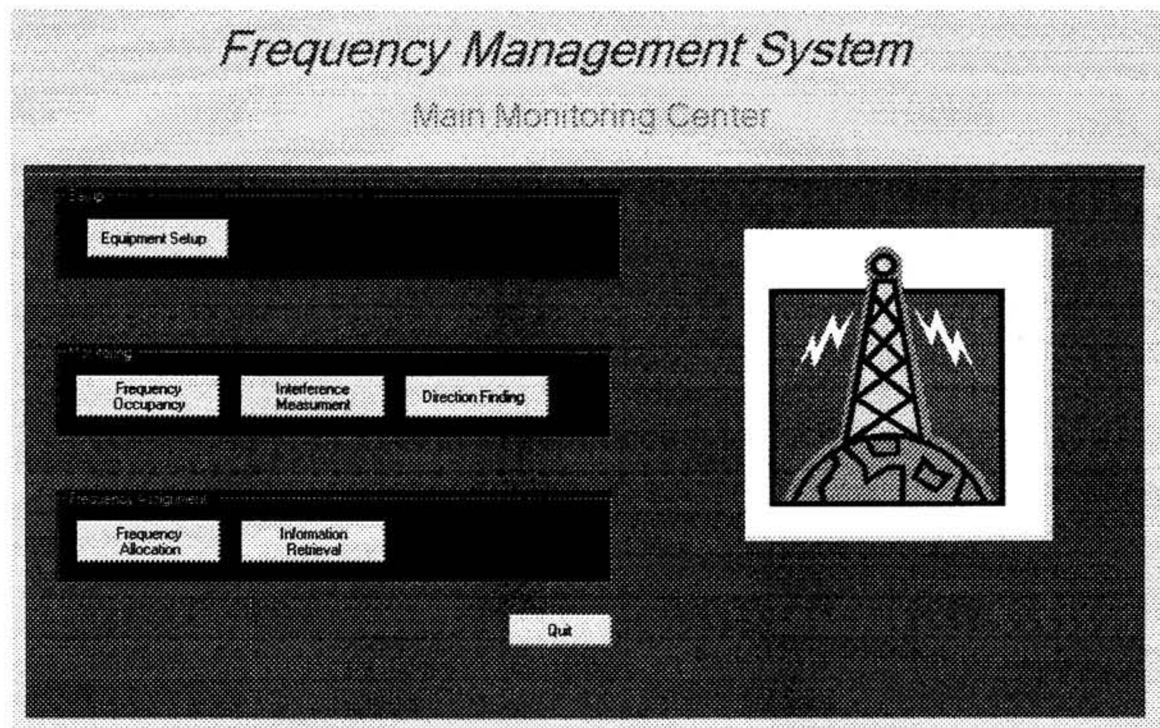


Figure A-4 User interface - MainUI

Activity modeling

Figure A-5 shows all possible activities occurring at the "MainUI" user interface, including optional and alternative activities. Feature conditions are used to define optional and alternative paths to the customizable workflow. The Equipment Setup button of Figure A-4 is used to invoke one of the following alternative user interfaces based on the customization process of target application: "MMCconnect", "RMSconnect", or "MMSconnect" user interfaces. "FreqOccupancy", "InfoRetrieval", "DirectionFind" and "FrequencyAlloc" are optional user interfaces. The buttons related to the invocation of these optional user interfaces are either enabled or disabled based on feature selection during the customization process of target applications.

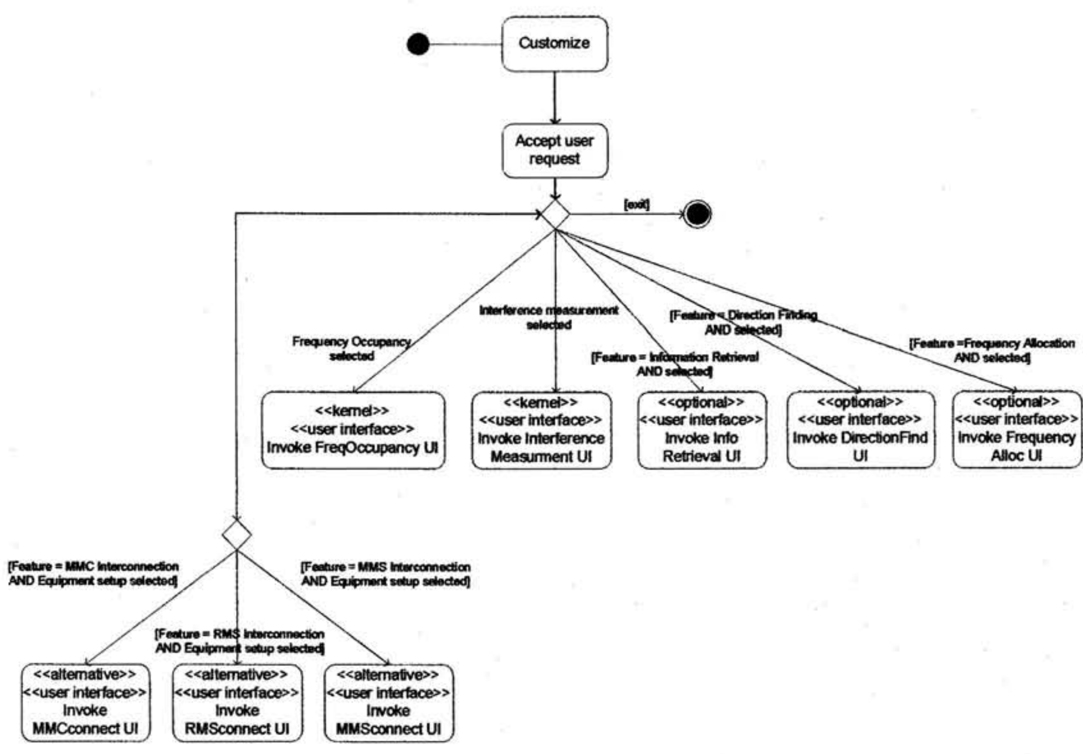


Figure A-5 Activity diagram - MainUI user interface

Interaction Modeling

Figure A-6 shows the customization phase of the "MainUI" user interface for the Dynamic Customization of Client Application (DCAC) and the Dynamic Customization of Client Application with Separation of Concerns (DCAC-SC) approaches, in which customization is done at run time by reading the selected features and parameterized variables from the customizer object.

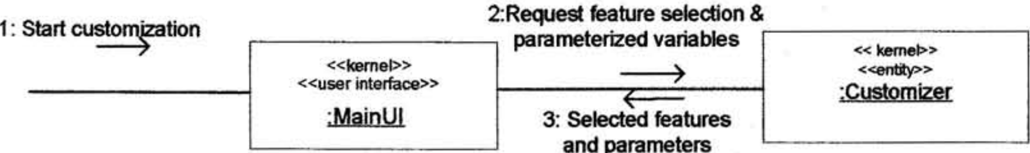


Figure A-6 Customization Phase - MainUI user interface

Figure A-7 shows the object interaction of "MainUI" user interface. Object interaction is based on the activity model, shown in figure A-5, and the description for that model.

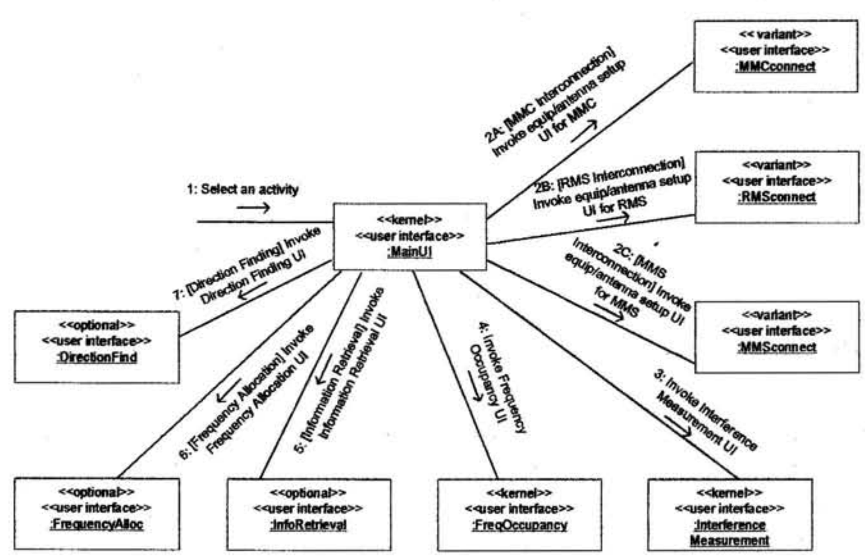


Figure A-7 Interaction Modeling - MainUI user interface

A.3.4.2 Equipment/Antenna Setup

The Radio Frequency Management System is created to support three different types of monitoring stations: Main Monitoring Center (MMC), Regional Monitoring Station (RMS), and Mobile Monitoring Station (MMS). Each station type contains different equipment and antenna setup. Therefore, three alternative user interfaces are developed to support the automatic setting of monitoring equipments and their related antennas of each station type. Figure A-8 shows the alternative "MMCconnect" user interface used at the Main Monitoring Center.

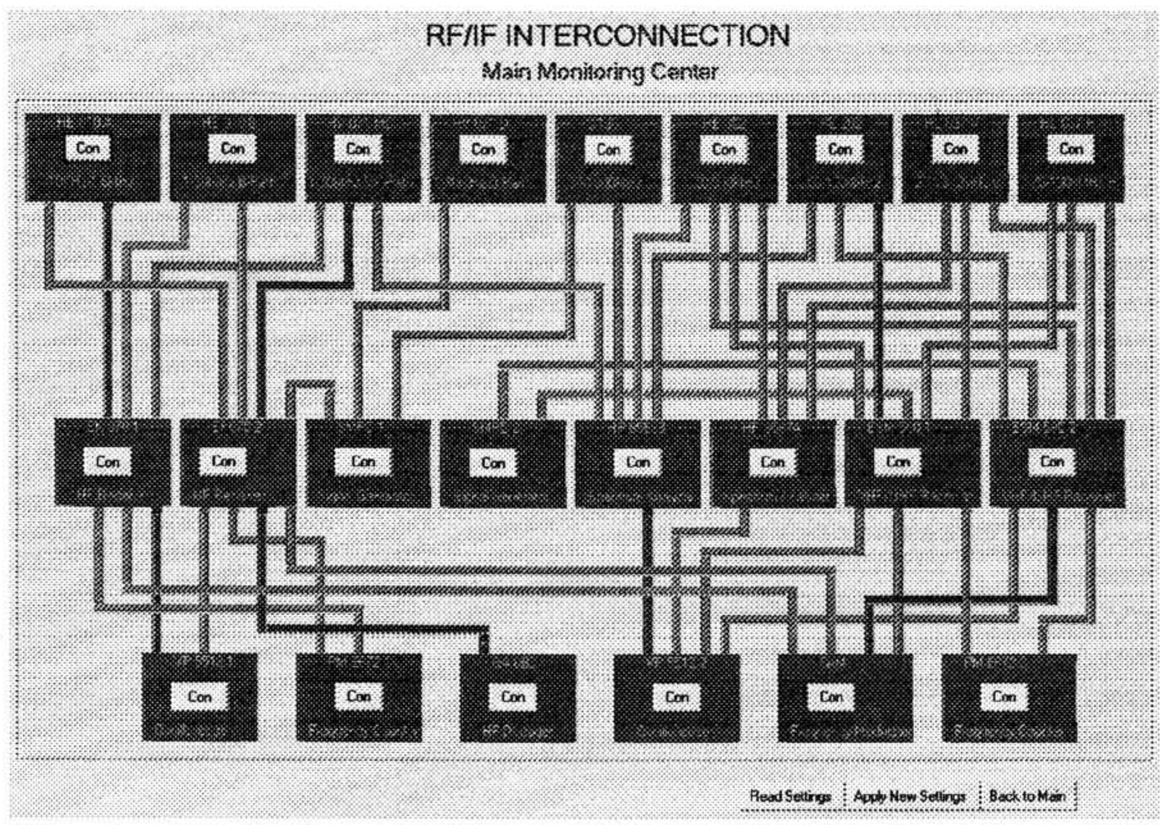


Figure A-8 Equipement/Antenna setup for MMC

Activity Modeling

Figure A-9 shows the activity diagram for "MMCconnect" user interface. It shows two possible web service invocations. The first invocation is for reading the current equipment/antenna interconnection setup for the Main Monitoring Center and the second invocation is for setting the new interconnection changes to the equipment/antenna setup.

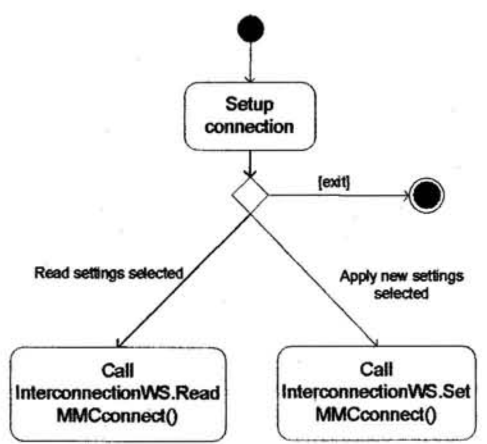


Figure A-9 Activity Diagram – MMCconnect UI

Interaction Modeling

Figure A-10 shows the object interaction for "MMCconnect" user interface of the Main Monitoring Center. The user interface has two functions: read the current equipment/antenna interconnection and set the new interconnection setup for the Main Monitoring Center.

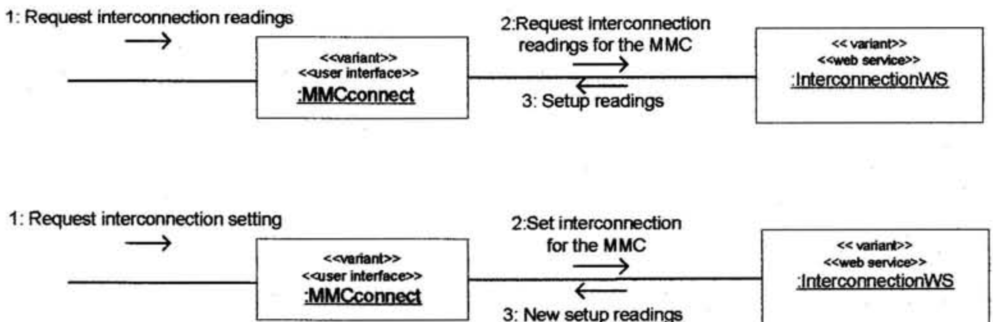


Figure A-10 Collaboration Diagram - MMCconnect UI

The regional and mobile monitoring stations have different equipment/antenna types and setup. Therefore, reading the equipment/antenna setup and setting the connection between the equipment and antennas require different user interfaces and web service methods to accomplish the above tasks. The different readings and settings of equipment/antenna web service methods are grouped in the InterconnectionWS web service, as shown in Figure A-11. Web service methods are depicted from the activity model of each user interface.

<<variant>> <<web service>> InterconnectionWS
ReadMMCconnect() ReadRMSconnect() ReadMMSconnect() SetMMCconnect() SetRMSconnect() SetMMSconnect() RotateD130() RotateHL023()

Figure A-11 InterconnectionWS

Figure A-12 shows the alternative "RMSconnect" user interface. It is used to perform automatic setting of monitoring equipments and their related antennas at the Regional Monitoring Stations (RMS).

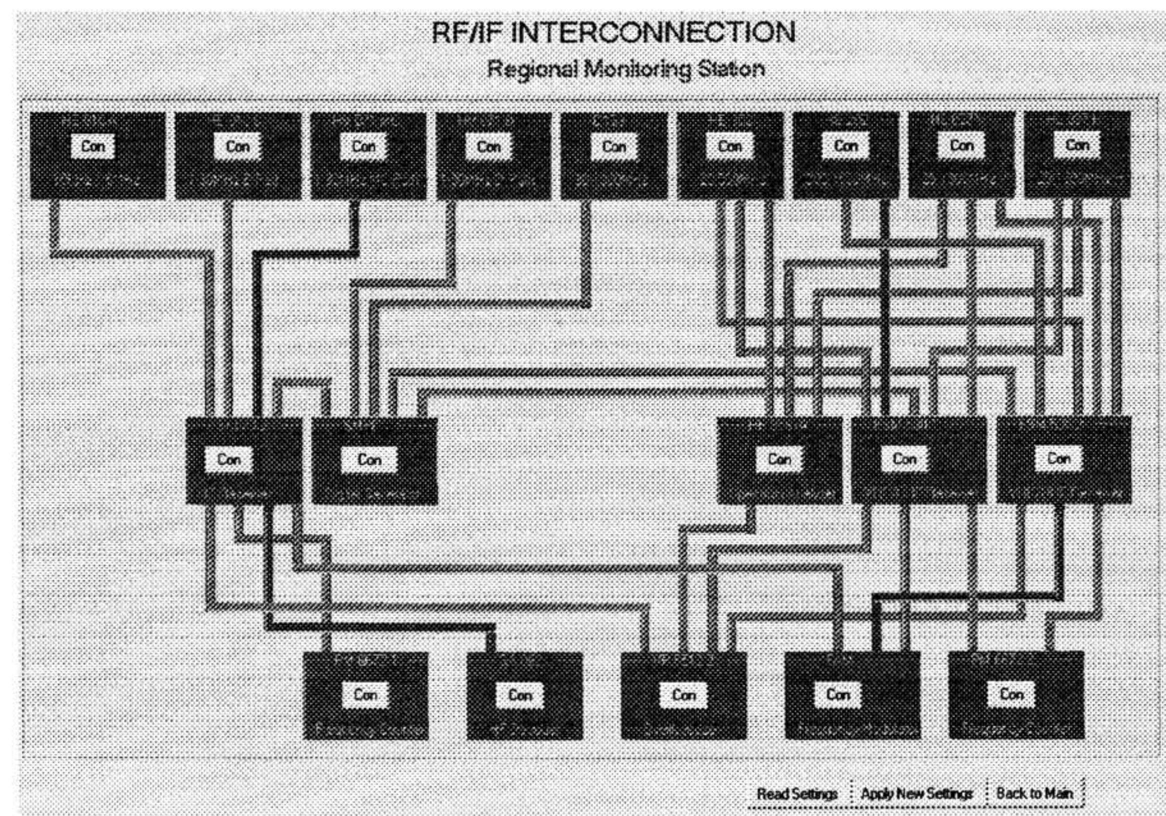


Figure A-12 Equipment/Antenna Setup - RMS

Activity Modeling

Figure A-13 shows the activity diagram for "RMSconnect" user interface. It shows two possible web service invocations. The first invocation is for reading the current equipment/antenna interconnection setup for the Regional Monitoring Station and the second invocation is for setting the new interconnection changes to the equipment/antenna setup.

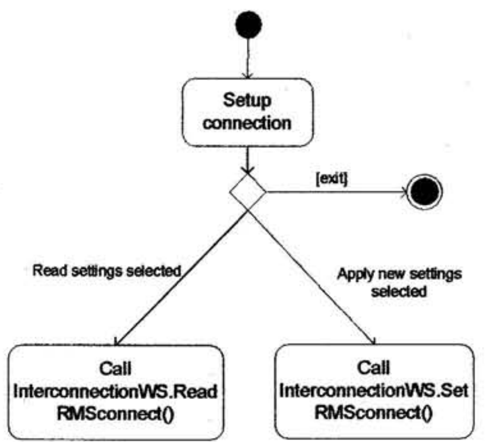


Figure A-13 Activity Diagram – RMSconnect UI

Interaction Modeling

Figure A-14 shows the object interaction for "RMSconnect" user interface. The user interface has two functions: read the current equipment/antenna interconnection and set the new interconnection setup for the Regional Monitoring Station.

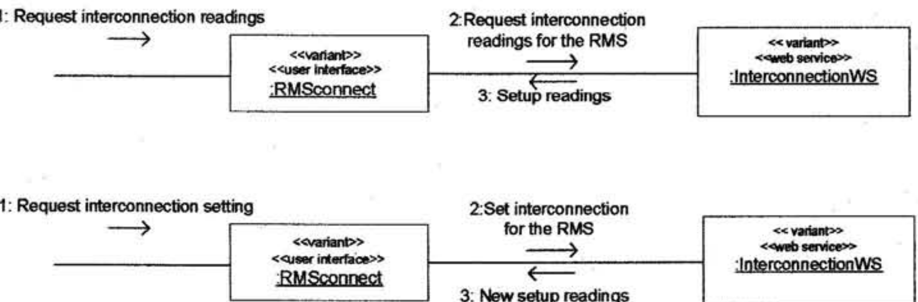


Figure A-14 Collaboration Diagram - RMSconnect UI

Figure A-15 shows the alternative "MMSconnect" user interface. It is used to perform automatic setting of monitoring equipments and their related antennas at the Mobile Monitoring Stations (MMS).

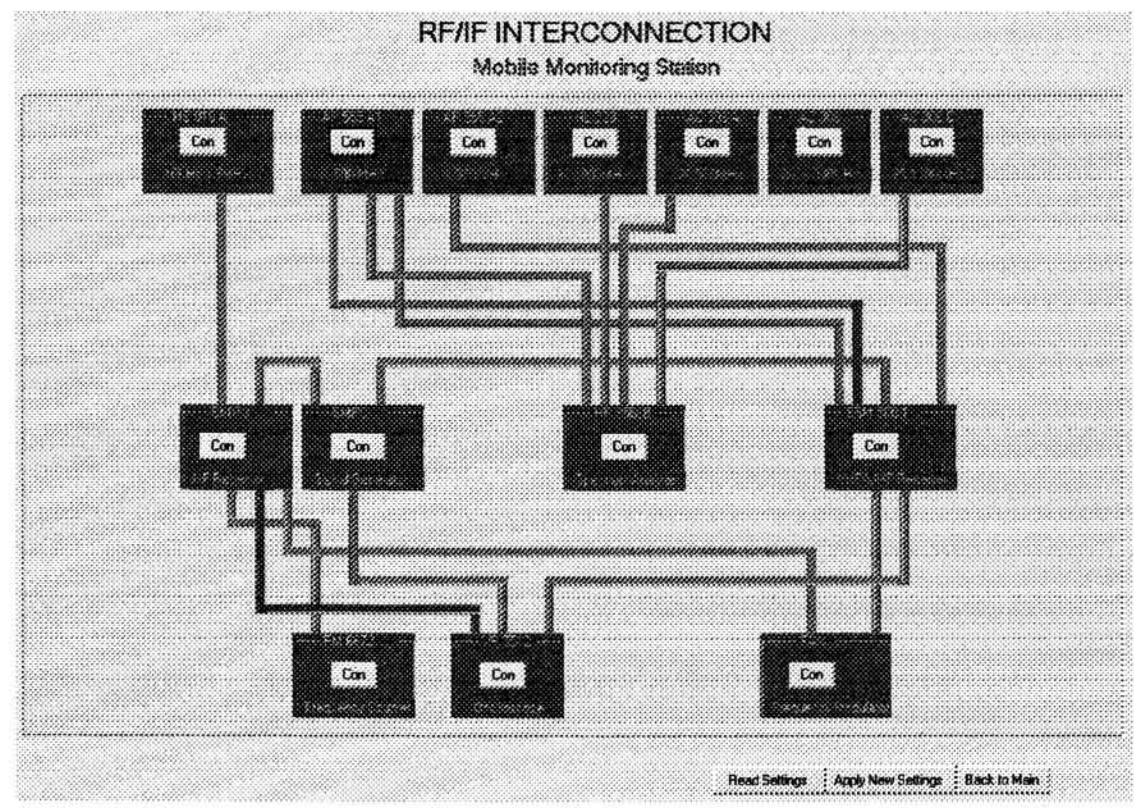


Figure A-15 Equipment/Antenna Setup - MMS

Activity Modeling

Figure A-16 shows the activity diagram for "MMSconnect" user interface. It shows two possible web service invocations. The first invocation is for reading the current equipment/antenna interconnection setup for the Mobile Monitoring Station and the second invocation is for setting the new interconnection changes to the equipment/antenna setup.

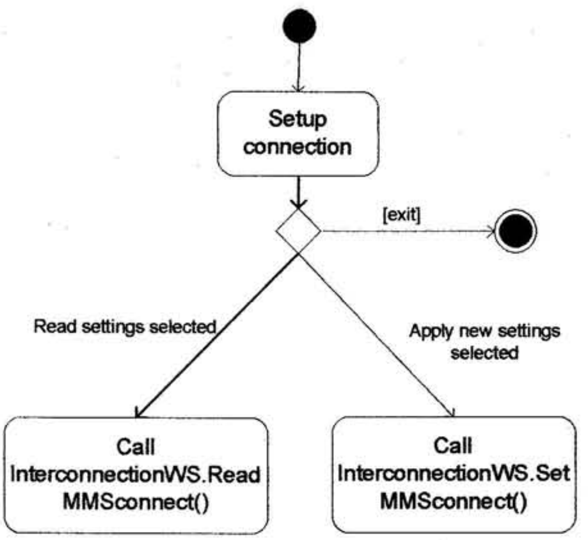


Figure A-16 Activity Diagram – MMSconnect UI

Interaction Modeling

Figure A-17 shows the object interaction for "MMSconnect" user interface. The user interface has two functions: read the current equipment/antenna interconnection and set the new interconnection setup for the Mobile Monitoring Station.

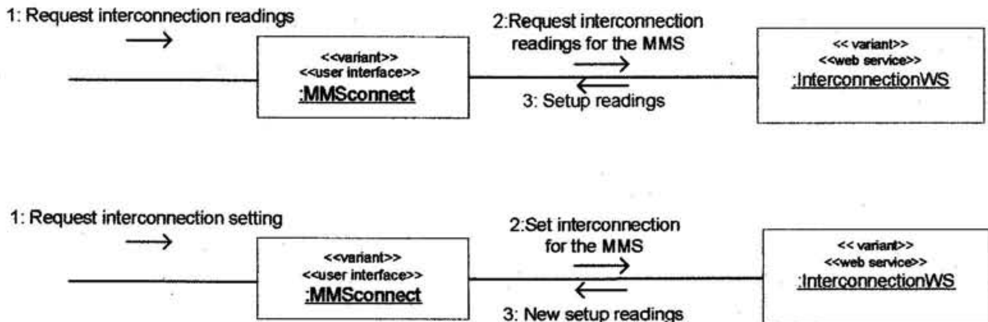


Figure A-17 Collaboration Diagram - MMS

A.3.4.3 Interference Measurement

The Interference Measurement user interface is used to process interference complaints by performing interference measurement tests on suspicious frequencies. Interferences are usually caused by illegal transmissions, malfunction of transmitters causing frequency deviation, or signal level increase. Figure A-18 shows the Interference Measurement user interface.

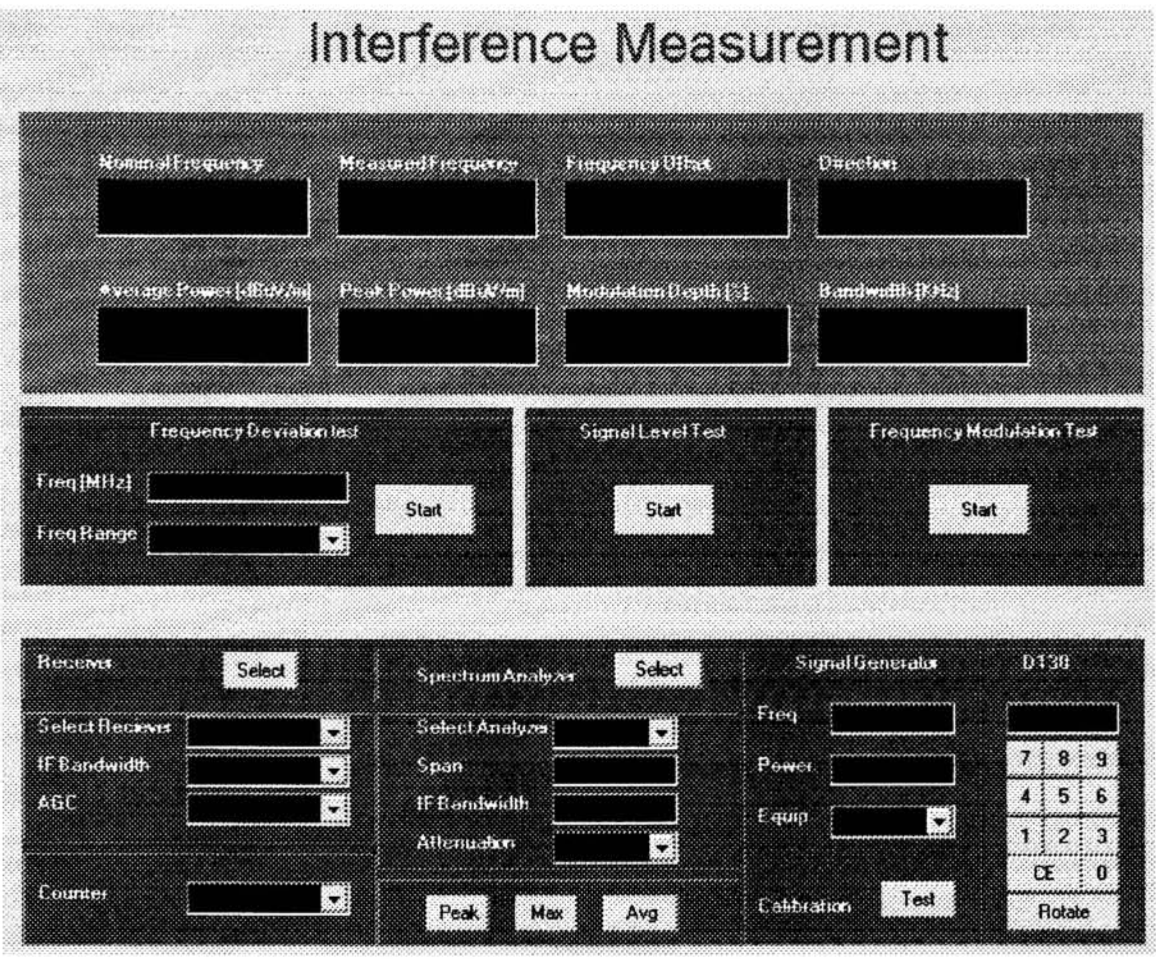


Figure A-18 Interference Measurement UI

Activity Modeling

The "InterferenceMeasurement" user interface is used to perform frequency interference calculations and run equipment calibration tests. The calculations are: frequency deviation, signal level, and frequency modulation. The equipment calibration function is used to test if measuring equipments used in the calculation tests are calibrated or need maintenance service. Also, this user interface allows users to rotate the appropriate antennas before running the measurement tests.

Figure A-19 shows the activity diagram for "InterferenceMeasurement" user interface.

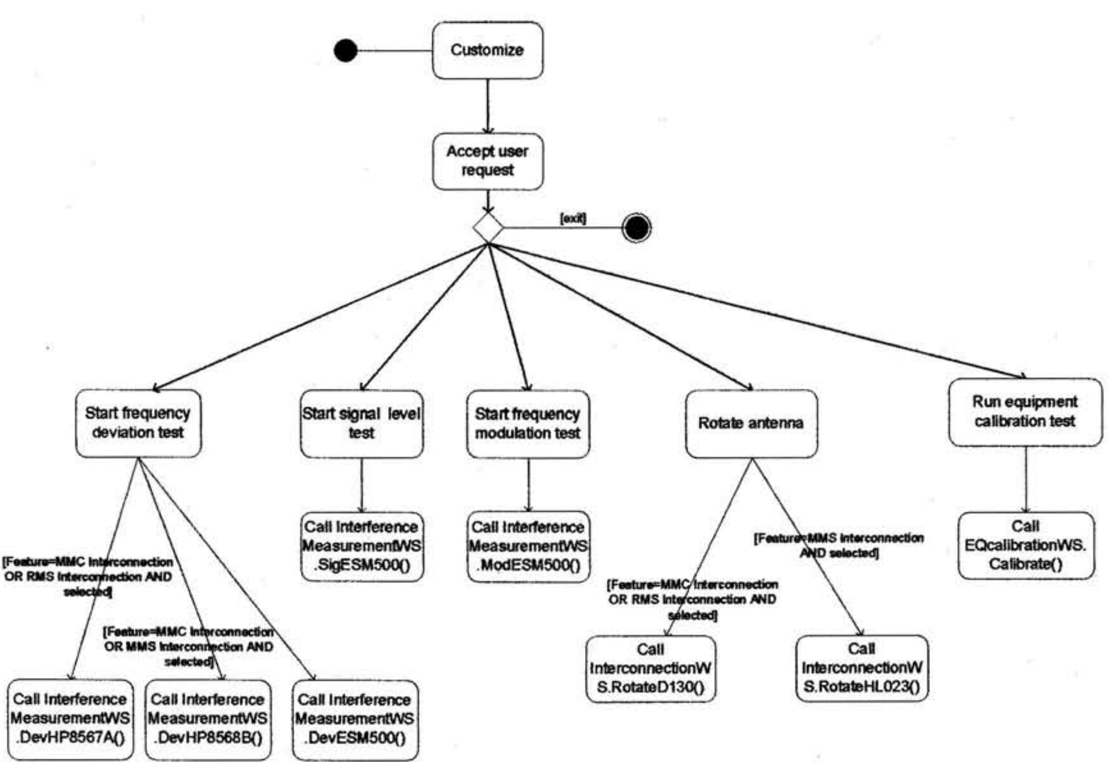


Figure A-19 Activity Diagram – InterferenceMeasurement UI

The above activity model shows the frequency deviation test is conducted using one of the following equipments: HP8587A spectrum analyzer, HP8588B spectrum analyzer, or ESM500 receiver. The feature conditions show that the spectrum analyzer HP8587A is associated with the MMC Interconnection and RMS Interconnection features. The spectrum analyzer HP858BA is associated with the MMC Interconnection and MMS Interconnection features. The ESM500 receiver is used by all type of stations.

The antenna rotation in the activity model shows two types of antennas: D130 and HL023. The feature conditions show that the D130 antenna is associated with the MMC Interconnection and RMS Interconnection features, while the D130 is associated with only the MMS Interconnection feature. Based on feature selection, the appropriate antenna is used in the antenna rotation activity.

Interaction Modeling

Figure A-20 shows the customization of the "InterferenceMeasurement" user interface for the DCAC and the DCAC-SC approaches, in which customization is done at run time by reading the selected features and parameterized variables from the customizer object.

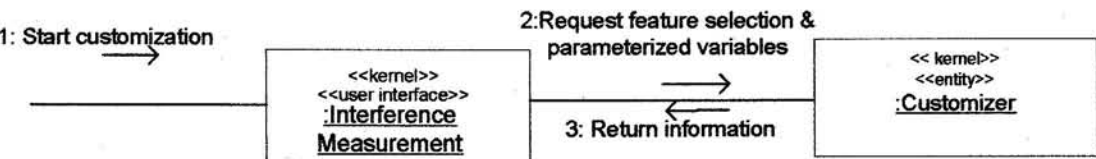


Figure A-20 Customization Phase – InterferenceMeasurement UI

Figure A-21 shows the collaboration diagrams for the "InterferenceMeasurement" user interface. The collaboration diagrams are based on the activity model of Figure A-19. They show the object interaction for the following activities: run deviation test, run modulation test, run signal level test, run calibration test, and rotate antenna. Also, web service objects are depicted from the activity model.

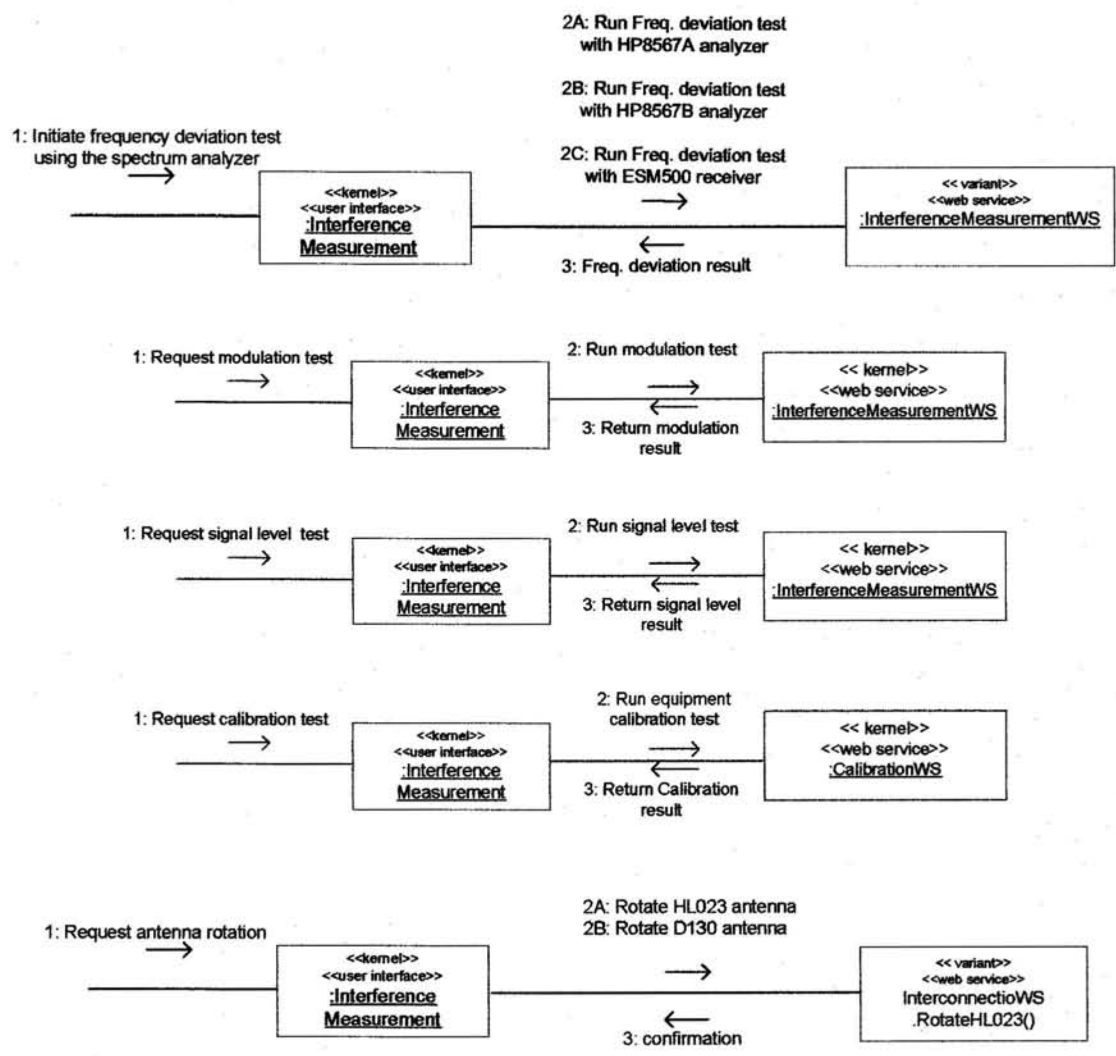


Figure A-21 Collaboration Diagram – Frequency Deviation

A.3.5 Web Services Modeling

From the activity modeling, all possible service requests are identified. These services are organized and grouped into related web services based on their objects interaction, described in section 3.4. Figure A-22 shows a sample grouping of methods into Web Services.

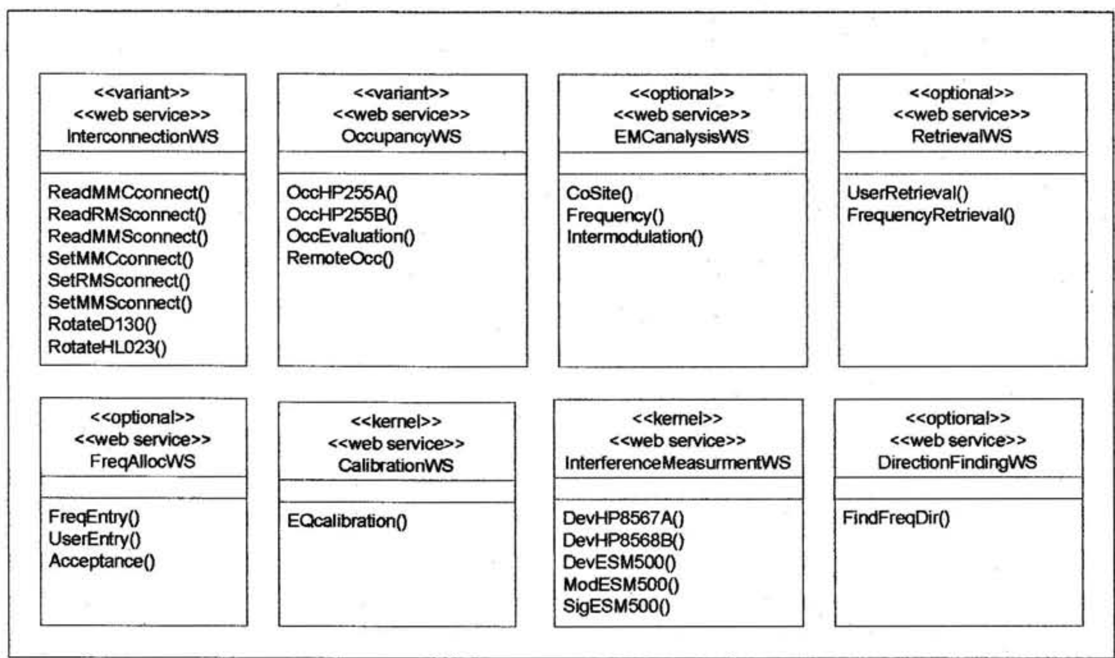


Figure A-22 Web Service Modeling

A.4. SPL development

This section applies the three development approaches for software product line based on web services to the Radio Frequency Management System (RFMS) case study. The "MainUI" user interface is used as an example for the development of each approach. This user interface object is customized to derive target applications using the SPLET tool. Figure A-23 shows the same activity diagram for the "MainUI" user interface described in section 3.4. The activity diagram is used to show all possible feature variation for derived applications. The feature guards are used in the development of the product line to apply customization decisions either during run time or during source code integration.

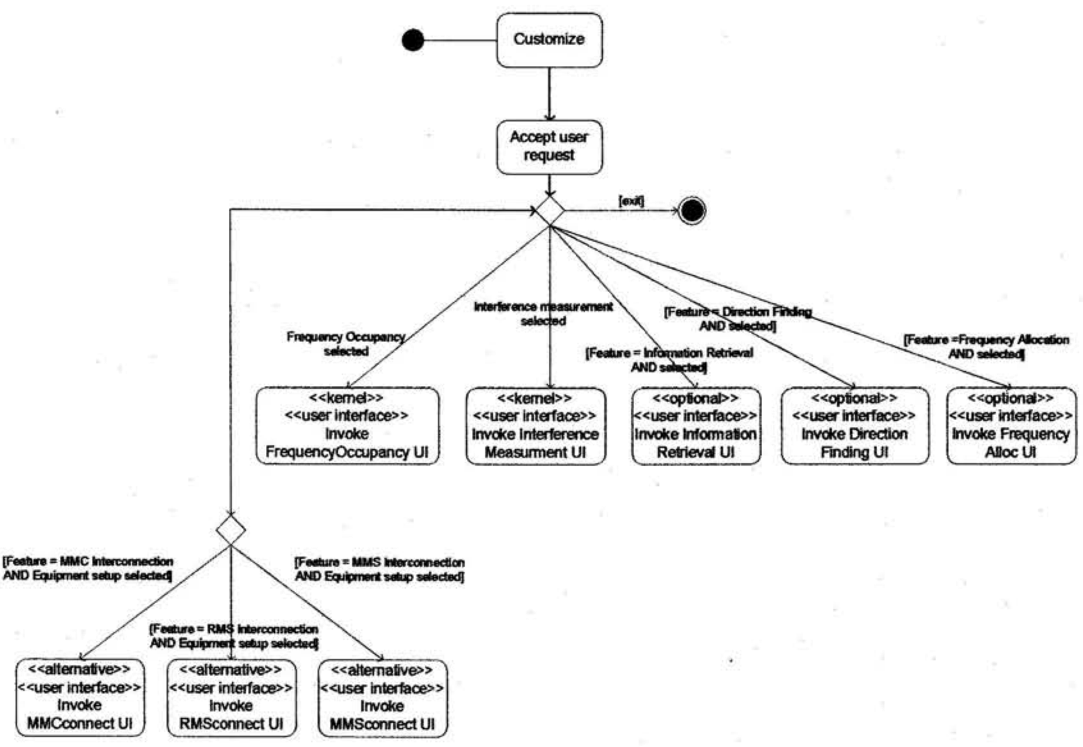


Figure A-23 Activity diagram - MainUI user interface

A.4.1 Dynamic Customization of Client Application (DCAC) approach

This section applies the DCAC approach to the RFMS case study, where target applications are dynamically customized at run time. Figure A-24 shows a sample implementation for the "MainUI" user interface. The source code sample shows how alternative and optional features are treated in the source code.

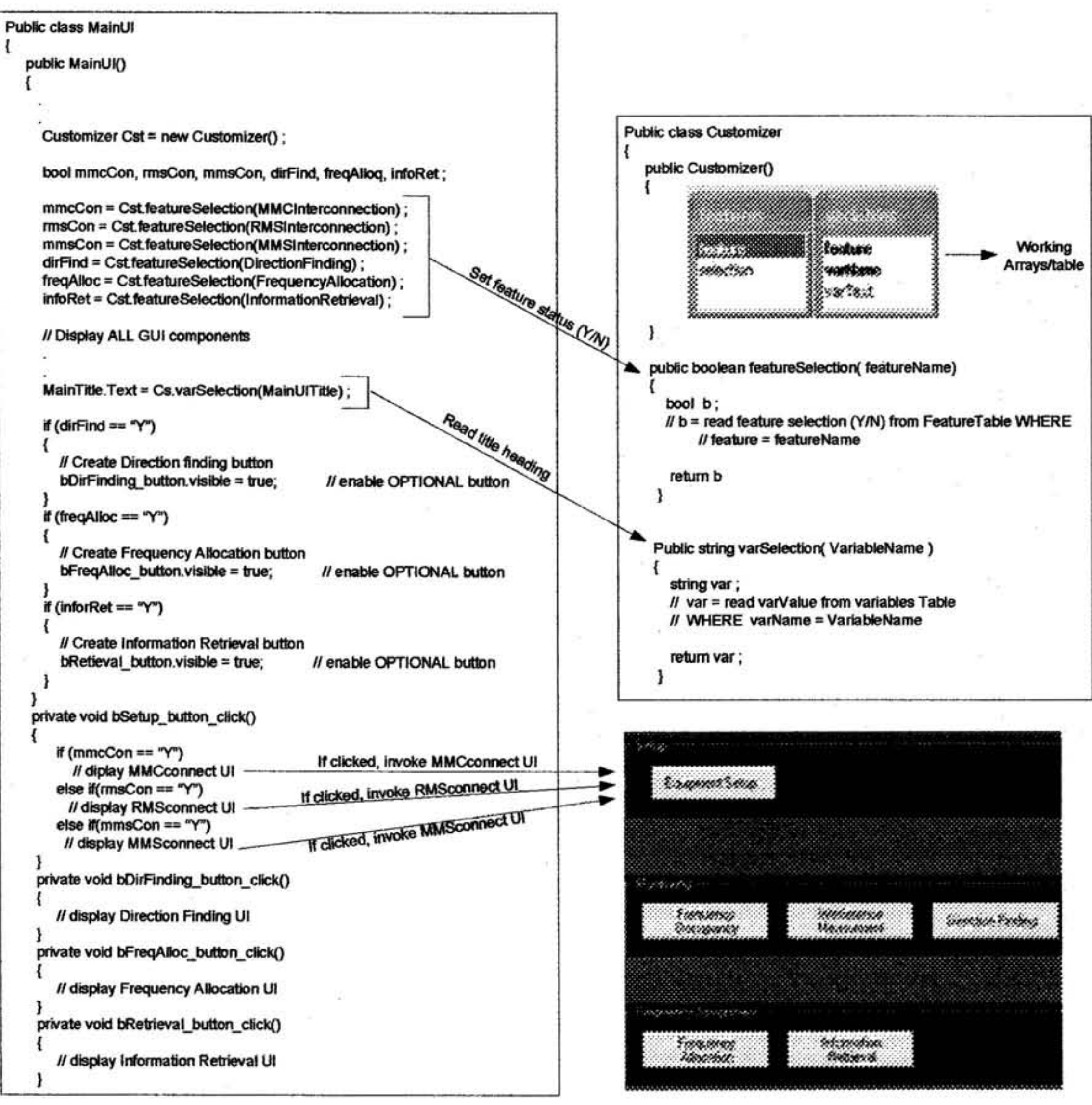


Figure A-24 DCAC Implementation - MainUI user interface

The "MainUI" user interface is customized by reading the feature selection and the value of parameterized variables from the customizer object to enable or disable buttons and set appropriate display variables. Its workflow is customized by setting features to true or false and applies settings to feature conditions on which user interface to call or which web service to invoke. The following section explains the customization in more detail.

A.4.1.1 Customization of client application at run time:

- Object MainUI is customized by reading the feature selections stored in the customizer object and stores them in local variables, where they will be used throughout the MainUI object. Local feature variables mmcCon, rmsCon, mmsCon, dirFind, freqAlloc, and infoRet store the MMC Interconnection, RMS Interconnection, MMS Interconnection, Direction Finding, Frequency Allocation, and Information Retrieval feature decisions respectively and are set to "Y" or "N", depending on whether the feature is selected or not.

- During the customization process, optional button "Direction Finding" is created if dirFind is equal to "Y" and ignored otherwise.

```
if (dirFind == "Y")
    // Create Direction Finding button
    bDirFinding_button.visible = true;
```

- During the customization process, optional button "Frequency Allocation" is created if freqAlloc is equal to "Y" and ignored otherwise.

```
if (freqAlloc == "Y")
    // Create Frequency Allocation button
    bFreqAlloc_button.visible = true;
```

- During the customization process, optional button "Information Retrieval" is created if infoRet is equal to "Y" and ignored otherwise.

```
if (infoRet == "Y")
    // Create Information Retrieval button
    bRetrieval_button.visible = true;
```

- During the customization process, the parameterized variable MainUITitle is read from the customizer object to set the appropriate header title of the "MainUI" user interface.

```
MainTitle.Text = Cst.varSelection(MainUITitle);
```

A.4.1.2 User interface object interaction:

After the dynamic customization process is complete, the "MainUI" user interface is ready to accept user input.

- If Equipment Setup button is invoked, "MMCconnect" UI, "RMSconnect" UI, or "MMSconnect" UI will be called, depending on whether MMC Interconnection, RMS Interconnection, or MMS Interconnection feature is selected.

```
if (mmcCon == "Y")
   // diplay MMCconnect UI
else if(rmsCon == "Y")
   // display RMSconnect UI
else if(mmsCon == "Y")
   // display MMSconnect UI
```

- If Direction Finding button is enabled and invoked, "DirectionFind" UI will be called.

```
private void blockRes_button_click()
{
        // display BlockReservation UI
}
```

- If Frequency Allocation button is enabled and invoked, "FrequencyAlloc" UI will be called.

```
private void bFreqAlloc_button_click()
{
        // display FrequencyAlloc UI
}
```

- If Information Retrieval button is enabled and invoked, "InfoRetrieval" UI will be called.

```
private void bRetrieval_button_click()
{
        // display InfoRetrieval UI
}
```

A.4.2 Dynamic Customization of Client Application with Separation of Concerns (DCAC-SC) approach

This section applies the DCAC-SC approach to the RFMS case study to include separation of concerns, which is not addressed in the DCAC approach. Figure A-25 shows a sample implementation for the "MainUI" user interface. The source code sample shows the separation of concerns between kernel source code and variable source code. The separated source code is then integrated with kernel source code during the code weaving process using the proof-of-concept SPLET environment. The result of the weaving process is the combined source code for the entire software product line including all optional and alternative source code. The code weaving process and compilation are performed only once to generate an executable SPL system containing all kernel and variable source code. Target systems will rely on the dynamic client application customization at system run time, the source code of which is identical to that produced by the first approach (DCAC).

The proof-of-concept prototype environment SPLET is used for separation of concerns and the integration of variable source code with kernel source code. Variable source code is created using the Variable Source Code Editor component of SPLET, described in Chapter 6 in section 6.2.3.1. The integration process is based on the dynamic method in the Code Weaver component of SPLET, described in Chapter 6 in section 6.2.3.3.

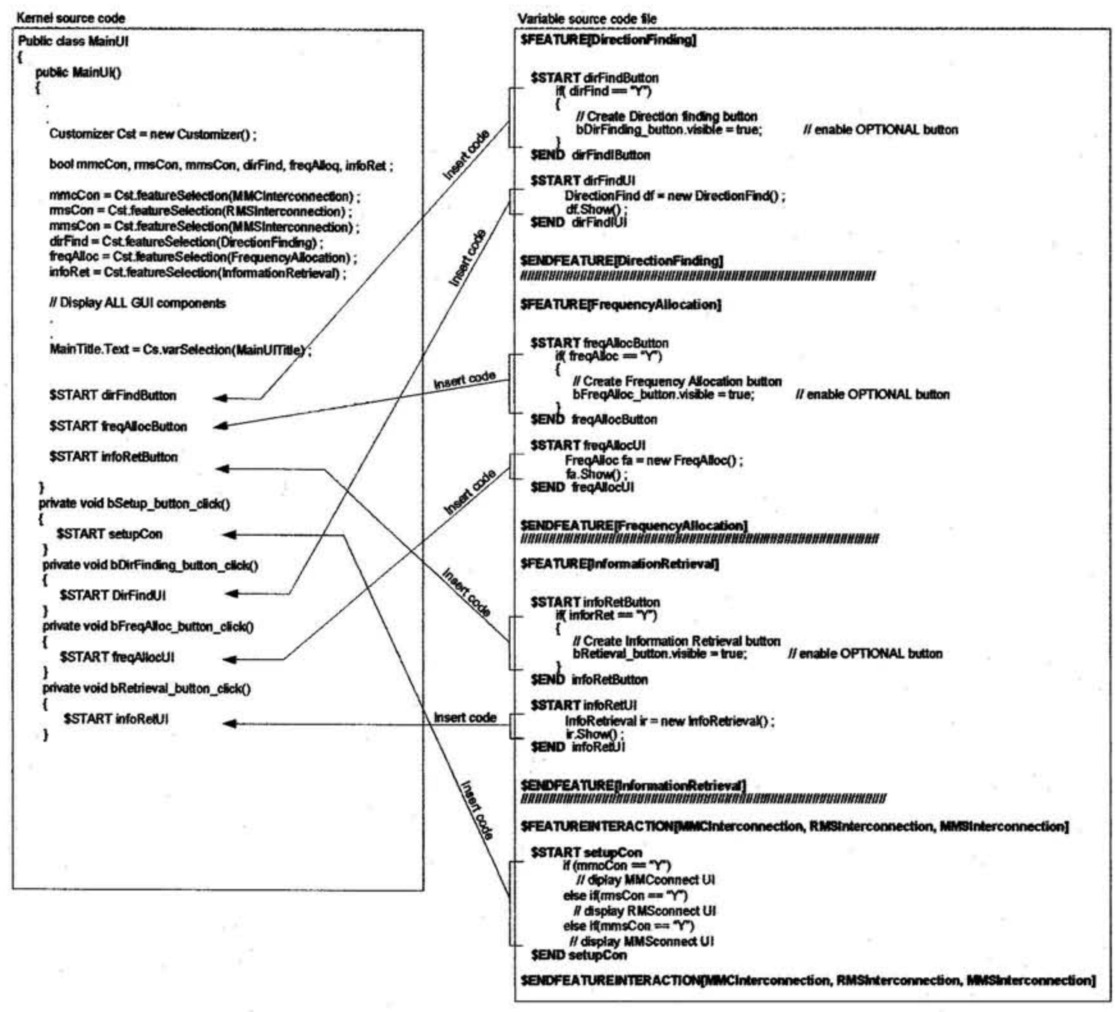


Figure A-25 DCAC-SC Implementation - Main Reservation UI

Based on the DCAC-SC approach, all optional and alternative feature source code in the variable source code file is integrated with the kernel source code at the location of the insertion point, using the dynamic method of integration in the Code Weaver component. For example, the insertion point *$START dirFindButton* refers to the optional feature "DirectionFinding" in the variable source code file. The variable source code will be inserted in the kernel "MainUI" user interface class at the place of the insertion point: *$START diFindButton*. At run time, this button will be either visible or invisible based on feature selection. The SPL application is customized at run time using a customization file that is produced by the Feature Selector, Consistency Checker, and Customization File Generator components of SPLET.

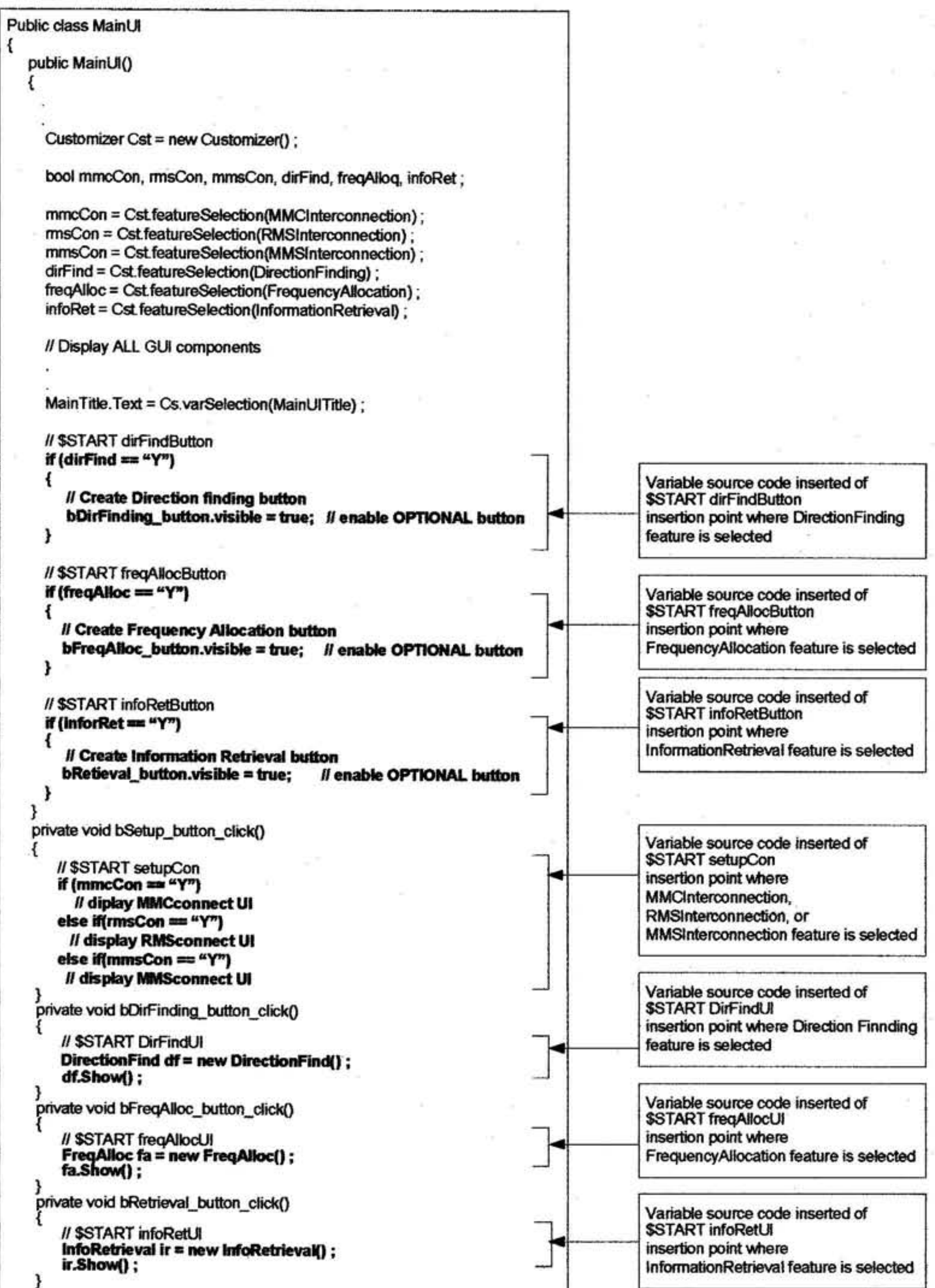


Figure A-26 Integrated Source Code - MainUI

Figure A-26 shows the "MainUI" user interface class after the integration process using the Code Weaver component. Inserted blocks are:

- Insertion point $START dirFindButton in the kernel source code is replaced with the following source code from the variable source code file:

```
// $START dirFindButton
if (dirFind == "Y")
{
    // Create Direction finding button
    bDirFinding_button.visible = true;  // enable OPTIONAL button
}
```

- Insertion point $START freqAllocButton in the kernel source code is replaced with the following source code from the variable source code file:

```
// $START freqAllocButton
    if (freqAlloc == "Y")
    {
      // Create Frequency Allocation button
      bFreqAlloc_button.visible = true;    // enable OPTIONAL button
    }
```

- Insertion point $START infoRetButton in the kernel source code is replaced with the following source code from the variable source code file:

```
// $START infoRetButton
    if (infoRet == "Y")
    {
        // Create Information Retrieval button
        bRetieval_button.visible = true;     // enable OPTIONAL button
    }
```

- Insertion point $START setupCon in the kernel source code is replaced with the following source code from the variable source code file:

```
// $START setuCon
if (mmcCon == "Y")
   // diplay MMCconnect UI
else if(rmsCon == "Y")
   // display RMSconnect UI
else if(mmsCon == "Y")
   // display MMSconnect UI
```

- Insertion point $START DirFindUI in the kernel source code is replaced with the following source code from the variable source code file:

```
// $START DirFindUI
DirectionFind df = new DirectionFind() ;
Df.show() ;
```

- Insertion point $START freqAllocUI in the kernel source code is replaced with the following source code from the variable source code file:

```
// $START freqAllocUI
FreqAlloc fa = new FreqAlloc() ;
fa.show() ;
```

- Insertion point $START infoRetUI in the kernel source code is replaced with the following source code from the variable source code file:

```
// $START infoRetUI
InfoRetrieval ir = new InfoRetrieval() ;
ir.show() ;
```

A.4.3 Static Customization of Client Application (SCAC) approach

This section applies the SCAC approach to the RFMS case study. In this approach, only source code related to selected features is integrated with kernel source code. Figure A-27 shows a sample implementation for the "MainUI" user interface. The source code sample shows both the kernel source code and optional and alternative source code in the variable source code file. Insertion points are the key for integrating kernel source code and variable source code. If an optional or an alternative feature is selected, its related source code from the variable source code file is inserted in the target application at the location of the insertion point.

The proof-of-concept prototype environment SPLET is used to create the separation of concerns and the integration of variable source code with kernel source code. Variable source code is created using the Variable Source Code Editor component of SPLET, described in Chapter 6 in section 6.2.3.1. The integration process is based on the static method in the Code Weaver component of SPLET, described in Chapter 6 in section 6.2.3.3.

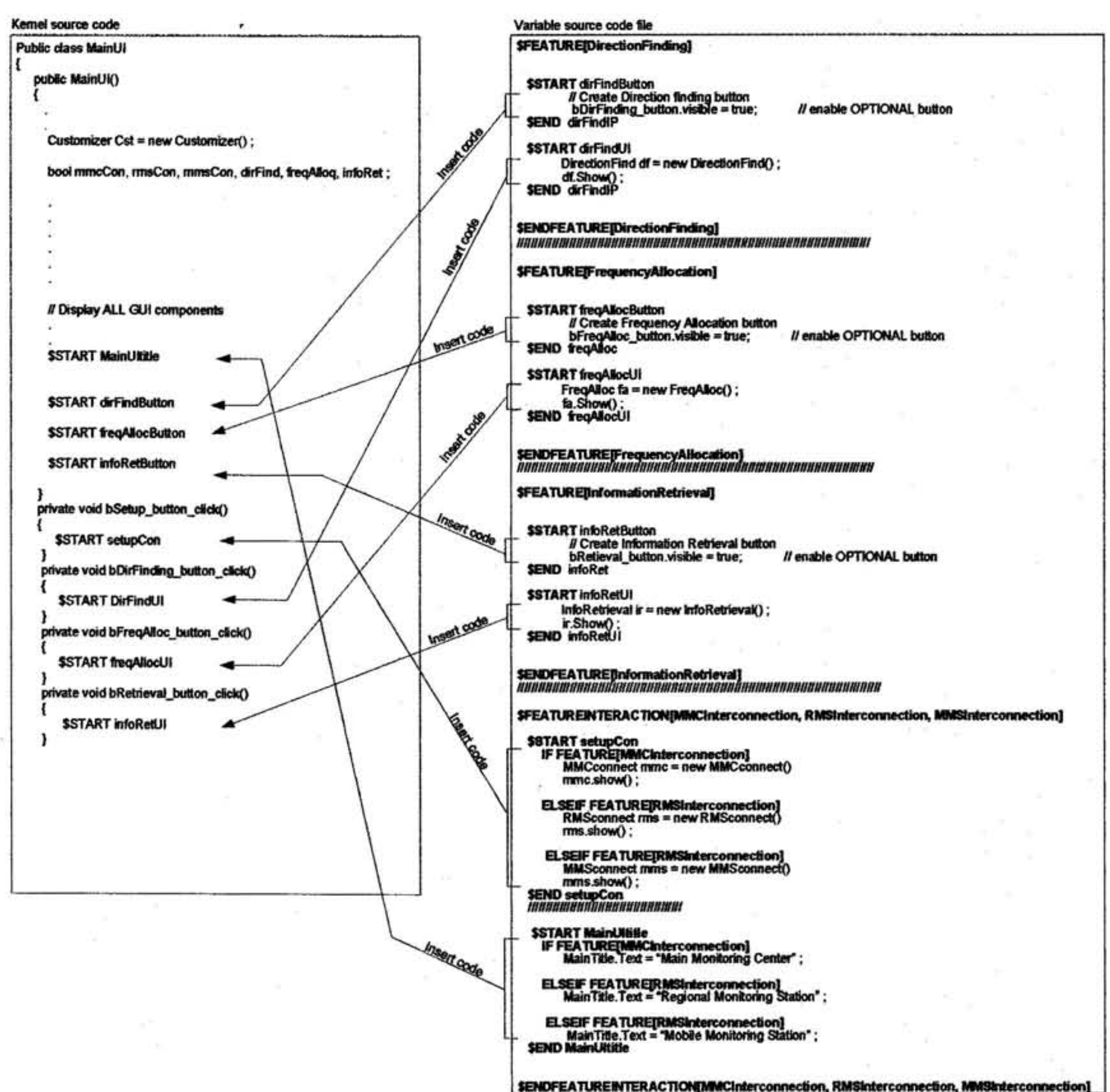


Figure A-27 SCAC Implementation - Main Reservation UI

Based on the SCAC approach, only selected optional and alternative source code from the variable source code file is integrated with the kernel source code at the location of the insertion point using the static method of integration in the Code Weaver component. For example, the insertion point *$START infoRetButton* refers to the optional feature

"InformationRetrieval" in the variable source code file. Only if this feature is selected using the Feature Selector component will the variable source code be inserted in the kernel "MainUI" user interface class at the place of the insertion point: *$STARTinfoRetButton.*

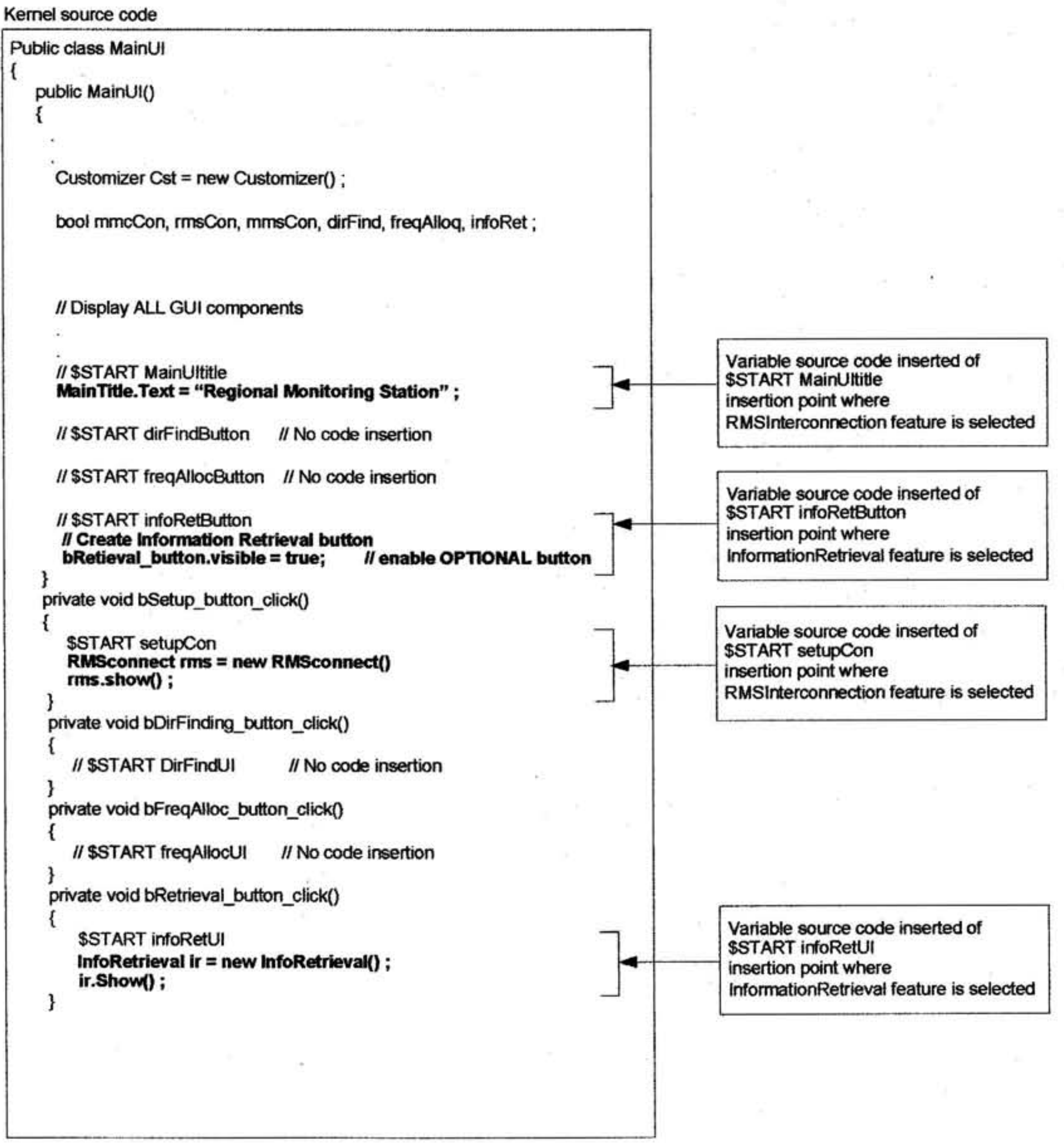


Figure A-28 Integrated Source Code - MainUI

Figure A-28 shows the "MainUI" user interface class after the integration process. In this example, the optional feature "InformationRetrieval" and the alternative feature "RMS Interconnection" are selected. The source code related to these feature is inserted in the kernel source code. The other features are not selected. Hence, their related source code is ignored during the integration process using the Code Weaver component. Inserted blocks are:

- Insertion point $START MainUItitle in the kernel source code is replaced by the following source code from the variable source code file:

```
// $START MainUItitle
MainTitle.Text = "Regional Monitoring Station" ;
```

- Insertion point $START InfoRetButton in the kernel source code is replaced by the following source code from the variable source code file:

```
// $START InfoRetButton
// CreateIinformation Retrieval Button
bRetrieval_button.visible = true ;
```

- Insertion point $START setupCon in the kernel source code is replaced by the following source code from the variable source code file:

```
// $START setupCon
RMSconnect rms = new RMSconnect();
Rms.Show() ;
```

- Insertion point $START infoRetUI in the kernel source code is replaced by the following code from the variable source code file:

```
// $START infoRetUI
InfoRetrieval ir = new InfoRetrieval();
ir.Show() ;
```

A.4.4 Summary

The Radio Frequency Management Systems is the second case study used to validate this research. This case study first modeled the multiple-views of the RFMS product line. Use case model, feature model, navigation model, GUIs, activity diagrams, and collaboration diagrams were used to design the RFMS product line. The design was then translated into implementation source code based on each of the three development approaches that are introduced in this research: Dynamic Client application Customization (DCAC), Dynamic Client application Customization with Separation of Concerns (DCAC-SC), and Static Client Application Customization (SCAC). The implementation source code of the three development approaches was customized to generate target applications from the product line.

Appendix B: Development Environment Patterns

B.1 Introduction

This chapter lists the patterns used in the three software development environments, described in sections 5.2, 5.4, and 5.5, to support the automatic customization of SPL architecture and components:

B2. Dynamic Client Application Customization (DCAC)

B3. Dynamic Client Application Customization with separation of concerns (DCAC-SC)

B4. Static Client Application Customization with separation of concerns (SCAC)

B.2 Dynamic Client Application Customization Pattern

Dynamic Client Application Customization Pattern

Intent

Provide a consistent reusable solution to the implementation architecture of a client/server software product line using web services with provision for *dynamic* client application customization.

Motivation

The goal of developing software product lines is to promote flexible software reuse. With the introduction of web services to SPLs, there is a need for developing a systematic approach that enables developers to implement a customizable system that can be dynamically customized into many single target systems without the need to modify any of the source code. Using the feature selector component, user interfaces and workflows of SPL systems can be automatically adjusted at *run time* to serve a single target system.

Solution

The idea behind the (DCAC) pattern is the development of dynamic client application that can be customized at system run time.

The DCAC Pattern has two main steps:

3. SPL Customization
4. Target application interaction

Step 1: SPL Customization

This step involves selecting desired optional and alternative features to be included in the target system. The feature selector component provides a facility to make feature selection from a SPL model and run consistency checks to verify selections. Once features are selected, selection information will be stored in the customization file by the customization file generator. The dynamic client application is customized by reading the customization file at run time.

(DCAC pattern – Continue)

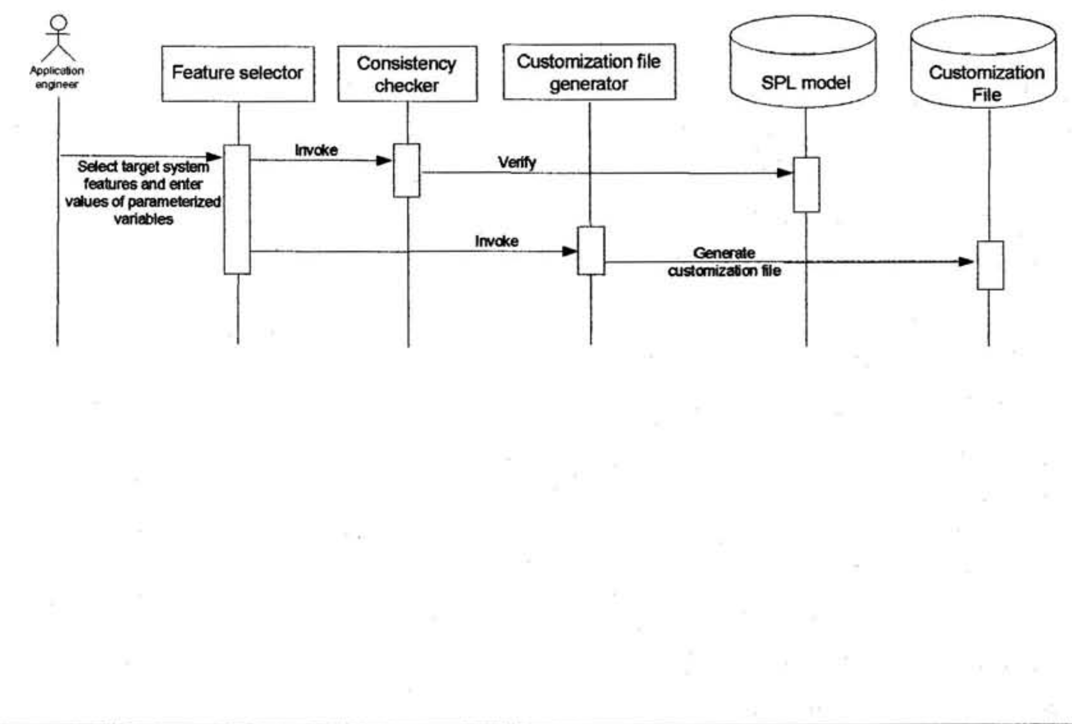
Components description:

- Feature selector: Allows users to selects desired features, and allows entry for parameterized variable values.
- Consistency checker: Verifies feature selection.
- Customization file generator: Generates a customization file for each target system.
- SPL model database: Contains feature tree, feature relations, analysis model, design model, components, and parameterized variables.
- Customization file: Contains feature name, feature selection status (true/false) and values of parameterized variables.

Dynamics

The following scenario depicts the customization process of a target system:

- Application engineer selects desired features for a target system using feature selector component.
- Consistency checker is invoked to verify selection by consulting the SPL model.
- Generate a customization file, which will be used by the client application for dynamic customization at run time.



(DCAC pattern – Continue)

Feature editor
1 Invoke ▶ 1
Consistency checker
Verify ▶
1 1
SPL model
1
Invoke ▶
1
Customization file generator
Generate ▶
1 1..*
Customization File

Step 2: Target application interaction
The Dynamic Client Application Customization (DCAC) Pattern divides an interactive application into three components:

- Customizer component
- User interface component
- Web Service component

Customizer component contains all customization information for a single target system. At run time, the customizer object reads the customization file and stores all customization information in the customizer object's local storage (arrays, data table, etc.) to be used for customizing the client application user interfaces and their workflows. Customization information consists of enabled or disabled features and parameterized variables.

User interface component is responsible for accepting input from users and allowing invocation of possible service requests. It involves the sequencing of web services invocation and handling of message communication based on the customizable workflow. It is also responsible for displaying results to users coming from the web service component.

Web Service component is a collection of functional methods that are packaged as a single unit and published in the Internet, Intranet, or Extranet in a private or public UDDI for use by other software programs, in this case the user interface component.

(DCAC pattern – Continue)

Class Customizer	Collaboration
Responsibility - Reads customization information from the customization file / database	- Customization file

Class Web service	Collaboration
Responsibility - Process a service request based on provided input - Returns results of processed requests	- User interface

Class User interface	Collaboration

Responsibility
- Calls customizer class to:
  - Enable or disable user interface components based on selected features
  - Customize user interface
  - Customize workflow by setting up appropriate method calls and calls to other user interfaces based on selected features
- Invoke and pass parameters to appropriate web service(s)
- Receives results from web service(s)
- Display information to the user
	- Customizer
	- Web service

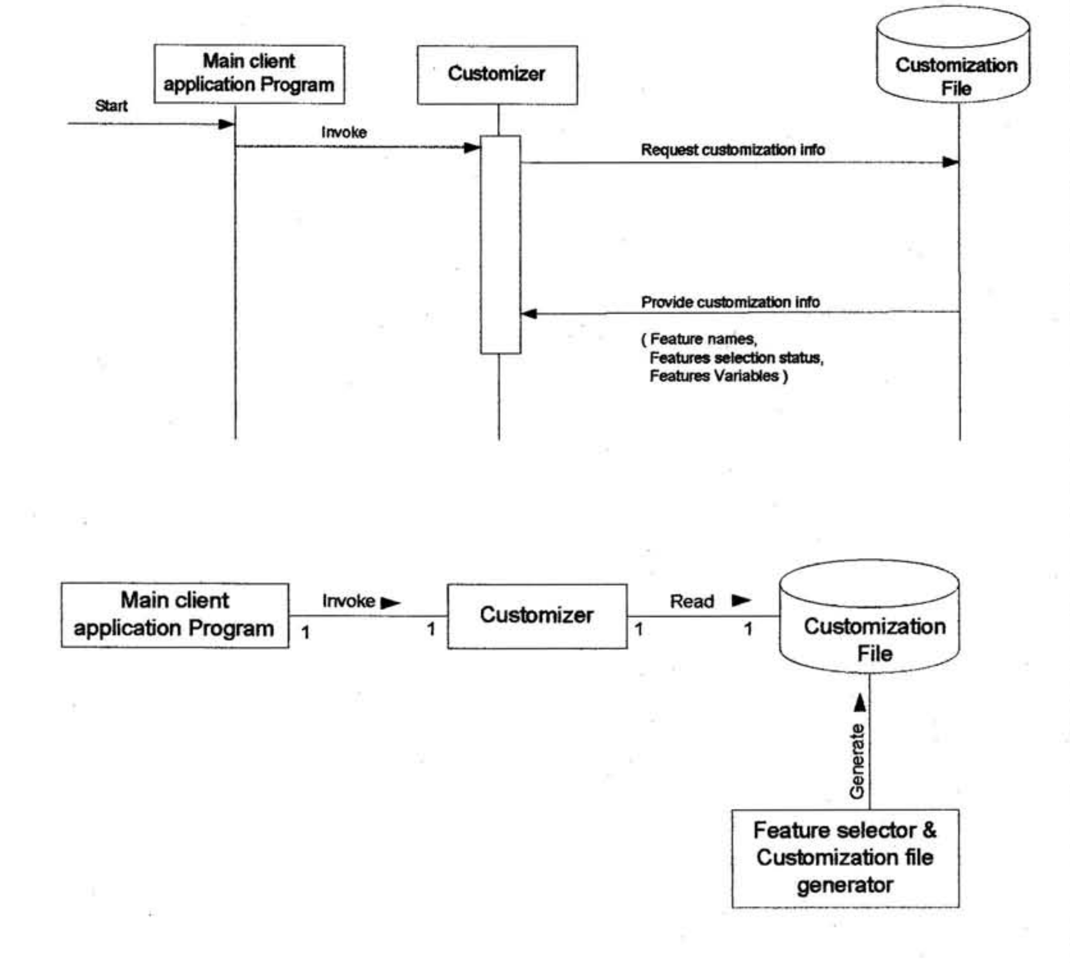
(DCAC pattern – Continue)

Dynamics

Once the target application features are selected in the SPL customization step, the application will be ready for execution. The application interaction step describes the two processes that occur at execution time: dynamic customization and object interactions.

Step 2-1: Shows how the client application is dynamically customized at run time.

- Starts main client application program.
- Customizer object is invoked at main client application program startup.
- Customizer object reads customization information once from the customization file that is generated by the customization file generator.
- Customization information can be read by all user interface objects through the customizer object.



(DCAC pattern – Continue)

Step 2-2: Shows how user interface objects interact with service requests using the DCAC pattern:

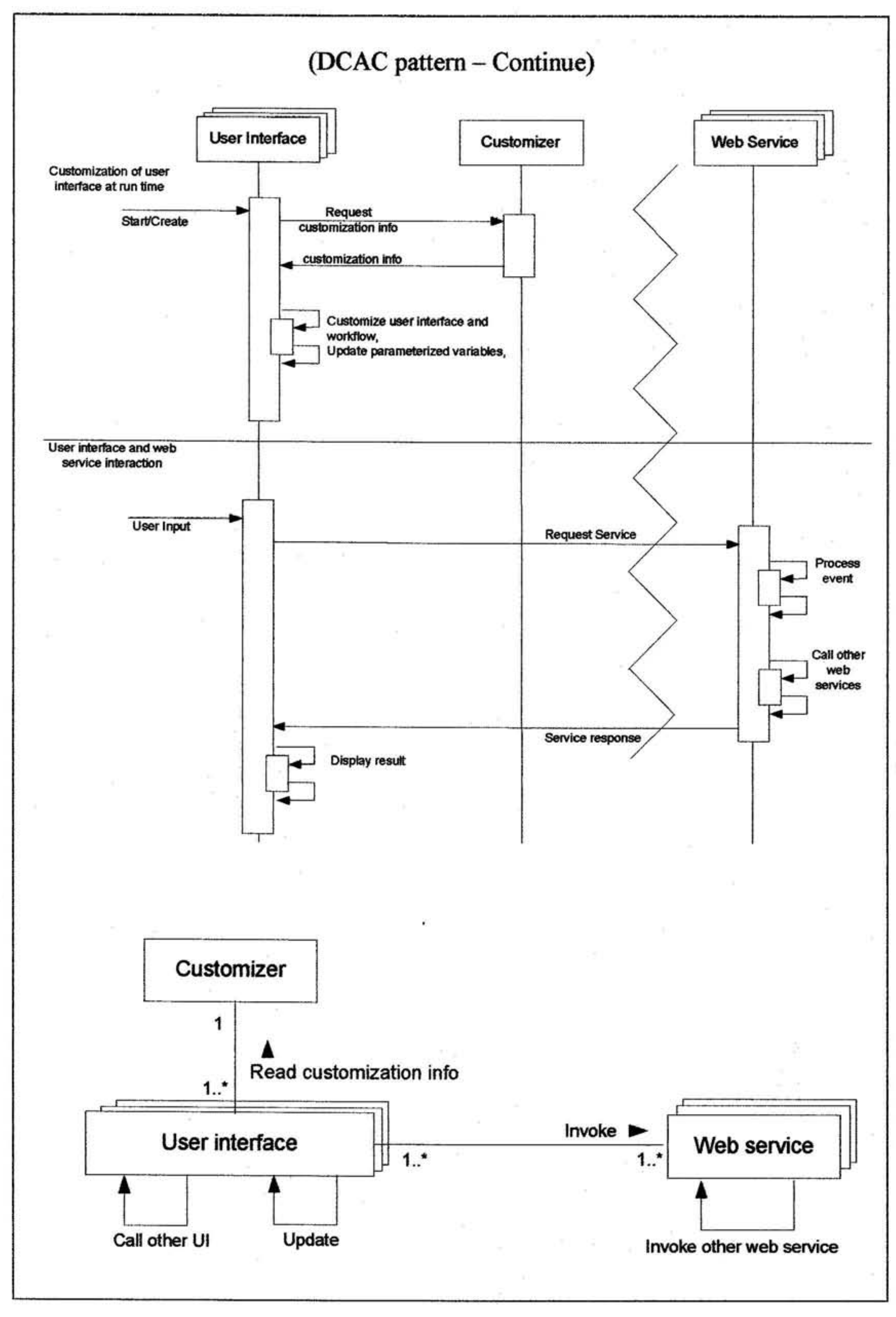
Customization of user interface at run time

- User invokes a user interface.
- User interface requests customization information from customizer object.
- User interface reads the customization information to:
    - Customize user interface components
    - Defining appropriate calls to web services based on selected features.
    - Define appropriate calls to other user interface objects.
    - Update parameterized variables.

  Customization is based on feature selection information stored in the customization file.

User interface and web service interaction

- User requests an activity by entering input data and clicking a button.
- User interface object passes the activity request and input data to a web service method(s).
- Web service processes the request and passes the results to the user interface object. A web service may also request services from other web services.
- User interface object displays results received from web service.


(DCAC pattern – Continue)
User Interface
Customizer
Web Service
Customization of user interface at run time
Start/Create
Request customization info
customization info
Customize user interface and workflow,
Update parameterized variables,
User interface and web service interaction
User Input
Request Service
Process event
Call other web services
Service response
Display result
Customizer
1
Read customization info
1..*
User interface
Invoke
1..*
1..*
Web service
Call other UI
Update
Invoke other web service

B.3 Dynamic Client Application Customization with Separation of Concerns Pattern

Dynamic Client Application Customization with Separation of Concerns Pattern

Intent
Provide a consistent reusable solution to the implementation architecture of a software product line using web services with provision for dynamic client application customization and separation concerns.

Motivation
This pattern is an extension to the DCAC pattern, which does not address the issue of separation of concerns. This issue needs to be introduced for the purpose of reducing complexity of developing SPL applications, maintenance, and system evolution.

Solution
The idea behind the (DCAC-SC) pattern is the development of dynamic client application that can be customized at system run time by separation of concerns between kernel source code and optional and alternative source code.

The DCAC-SC Pattern has four main steps:

5. Separation of concerns between kernel and variable source code
6. Code weaving
7. SPL Customization (the same as the DCAC pattern)
8. Target application interaction (the same as the DCAC pattern)

The above steps have to be performed in sequence. First, separation of concerns and code weaving have to be performed. The SPL application can then be customized by selecting desired features. Target applications are compiled to produce an executable SPL application.

(DCAC-SC pattern – Continue)

Step 1: Separation of concerns between kernel and variable source code:

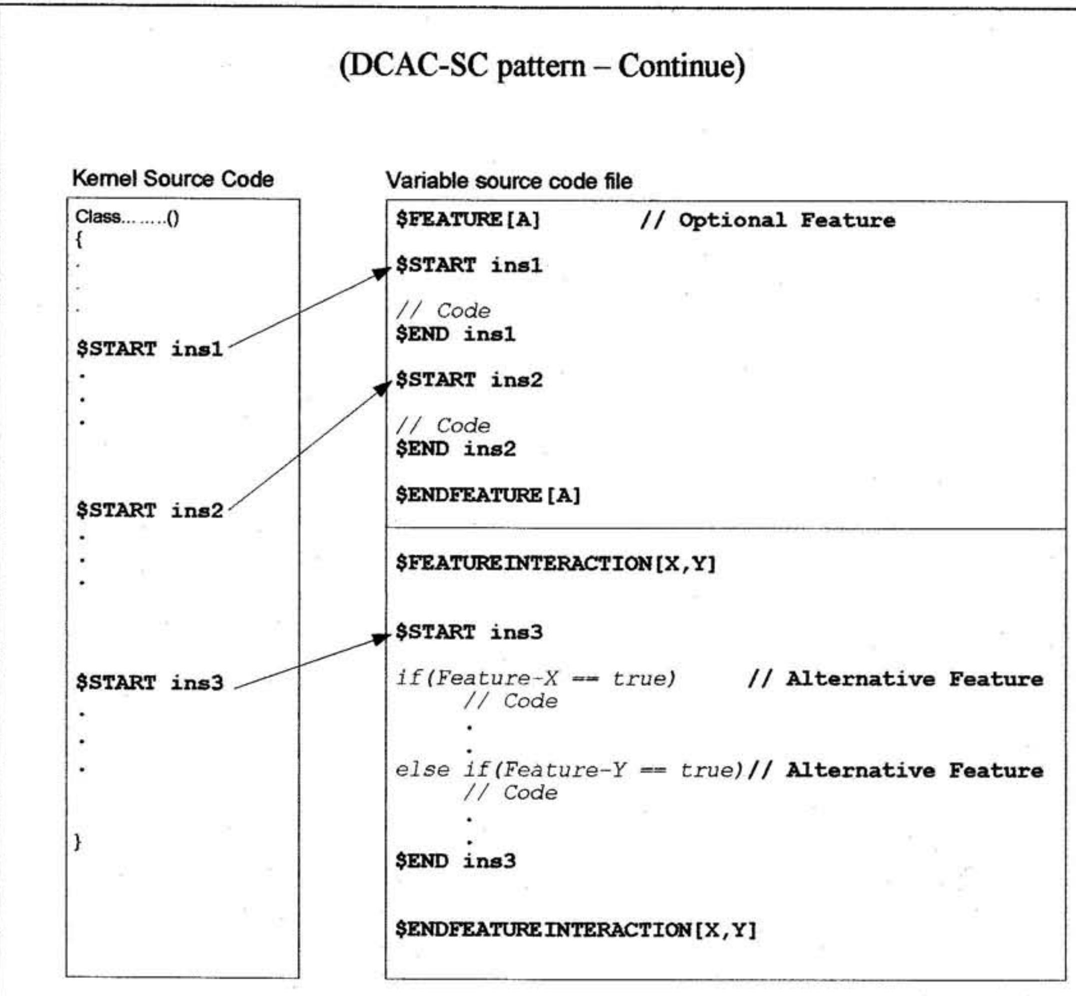
This step involves separating kernel source code from optional and alternative source code into a variable source code file where separated source code is grouped by features. Optional and alternative source code is identified by unique insertion point names in the variable source code file. Insertion points have to be also included in the kernel source code to specify the location where optional and alternative source code will be inserted.

Dynamics

The following scenario depicts the dynamic behavior of separation of concerns:

- Create application classes with kernel source code.
- Create a variable source code file that contains source code related to alternative and optional features.
- Add insertion points to kernel source code where optional and alternative source code from the variable source code file will be inserted.

(DCAC-SC pattern – Continue)



Language description:

- Kernel source code
  - $START <<insertion name>>: Specifies insertion location in kernel source code
- Variable source code file
  - $START <<insertion name>>: Identifies optional or alternative source code that needs to be inserted at the location specified in the kernel source code.
  - $END <<insertion name>>: Specifies the end of insertion code.

(DCAC-SC pattern – Continue)

- FEATURE [<<feature name>>]: Groups optional and alternative source code in a feature block. Feature blocks are integrated with kernel source code during the code weaving process based on insertion names.

- FEATUREINTERACTION[<<feature 1, feature 2, ...>>]: Groups related features source code that requires decisions on which source code to execute at run time. If-then-else statement is used within the insertion name of the feature interaction block with feature identifiers in the decision statement to be integrated as-is in the kernel source code based on the language used to develop the SPL application. At run time, only one of the decisions will be executed based on feature selection during SPL customization.

- ENDFEATUREINTERACTION []: Specifies the end of feature interaction code.

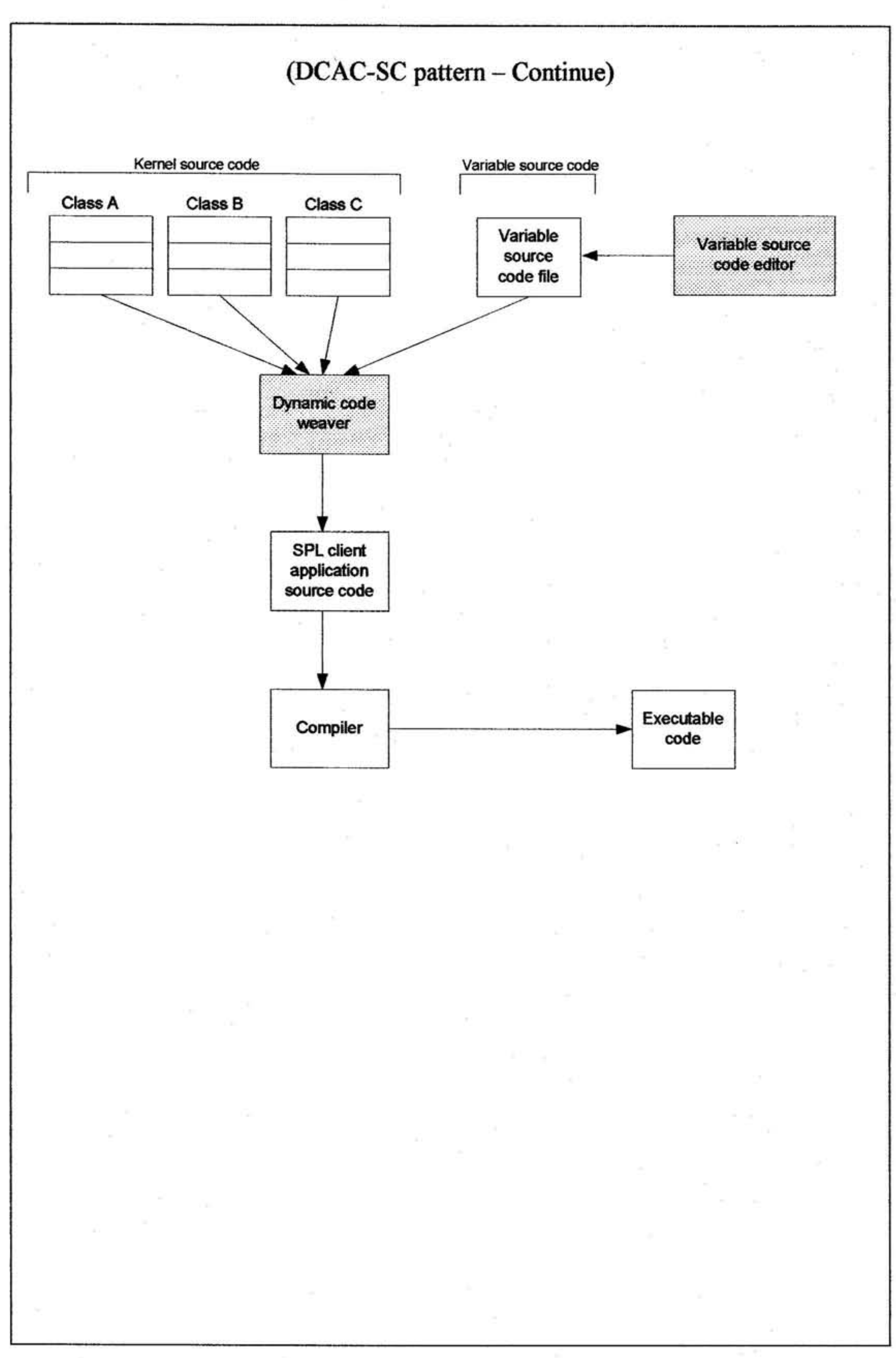
Step 2: Code weaving

This step combines kernel source code with optional and alternative source code from the variable source code file. This process is based on the Code Weaver component, which reads the variable source code file and inserts all source code blocks from that file into the kernel source code at the specified insertion locations.

Dynamics

The following scenario depicts the dynamic behavior of code weaving process:

- Run the code weaver component.
- Read optional and alternative source code from the variable source code file and integrate it into kernel classes at the specified insertion point locations.
- Compile integrated source code to generate an executable dynamic SPL application.


(DCAC-SC pattern – Continue)
Kernel source code
Variable source code
Class A
Class B
Class C
Variable source code file
Variable source code editor
Dynamic code weaver
SPL client application source code
Compiler
Executable code

(DCAC-SC pattern – Continue)

The following diagram shows the complete process of separation of concerns and source code integration:

Create Kernel code in classes

Develop client application classes

Kernel source code

Add Insertion points where all code from the feature file will be inserted

Add insertion points in kernel classes

Create a variable source code file that contains code related to alternative and optional features

Create variable source code file

variable source code file

Read variable source code file and integrate all source code into kernel classes at the specified insertion locations

Weave code

Kernel source code

Compile the integrated source code to generate an executable dynamic SPL system

Compile

Integrated source code

Run executable SPL application

Executable code

(DCAC-SC pattern – Continue)

Step 3: SPL Customization

This step is identical to the SPL customization step in the DCAC pattern. It involves selecting desired optional and alternative features to be included in the target application. The feature selector component provides a facility to make feature selection from a SPL model and run consistency checks to verify selections. Once features are selected, selection information will be stored in the customization file using the customization file generator. The dynamic client application is customized by reading the generated customization file at run time. This step is described in full in step 1 of the DCAC pattern.

Step 4: Target application interaction

This step is identical to the target application interaction step in the DCAC pattern. This step follows the SPL customization step. Once the target application features are selected, the application will be ready for execution. This step describes how the client application is customized dynamically at run time, and how user interface objects interact with service requests. This step is described in full in step 2 of the DCAC pattern.

B.4 Static Client Application Customization Pattern

Static Client Application Customization Pattern

Intent

Provide a consistent reusable solution to the implementation architecture of a software product line using web services with provision for *static* customization of client application using the concept of separation of concerns.

Motivation

The goal of developing software product lines is to promote flexible software reuse. With the introduction of web services to SPLs, there is a need for developing a systematic approach that enables developers to implement a customizable overall system that can be customized into many single target systems using a systematic method for extracting the required source code for each target system.

Solution

The idea behind the Static Client Application Customization (SCAC) pattern is the separation of concerns between kernel source code and optional and alternative source code for the purpose of extracting only required source code for running a target system.

The SCAC Pattern has four main processes:

5. Separation of concerns between kernel and variable source code
6. SPL Customization
7. Code weaving
8. Target system interaction

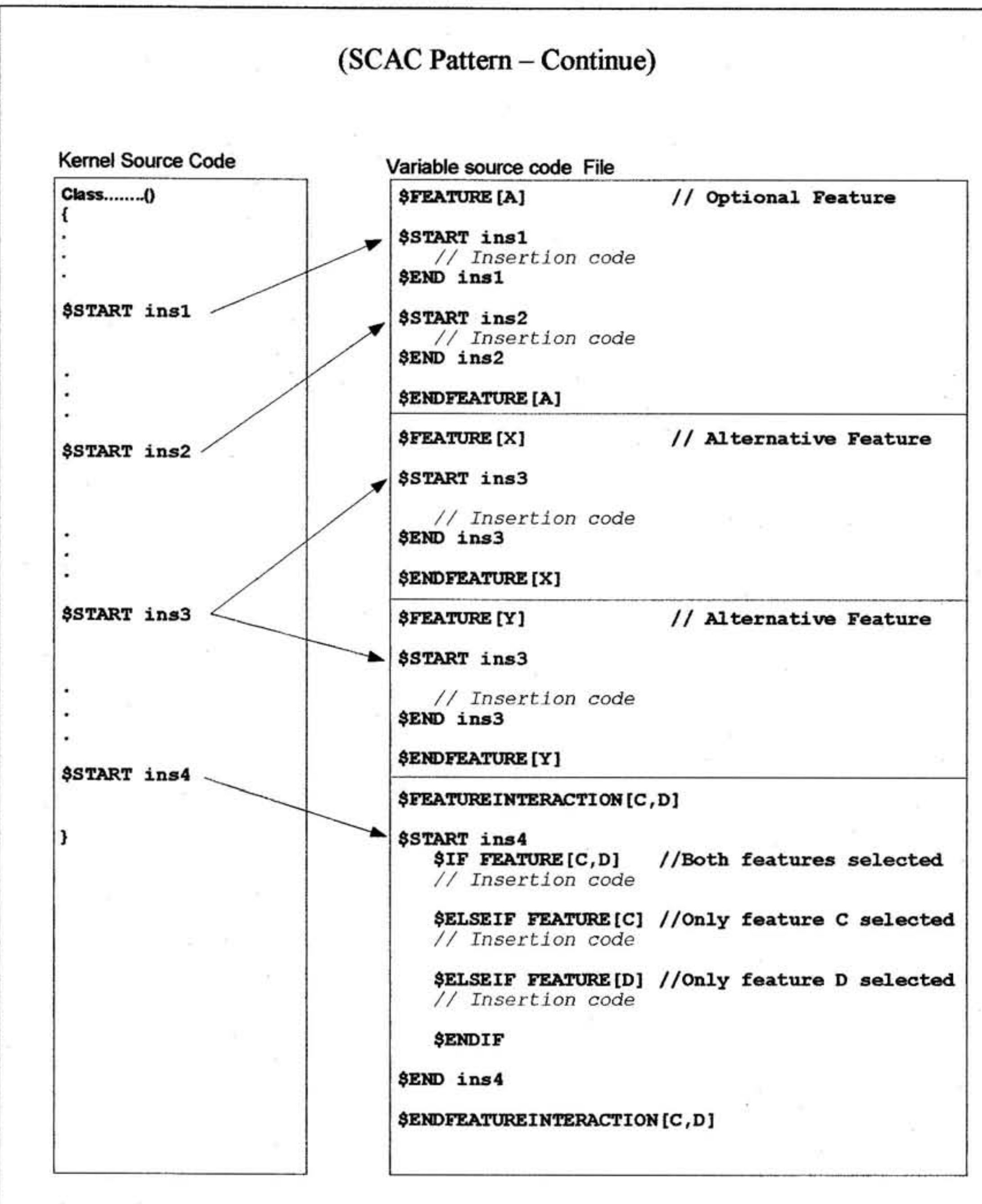
The above steps have to be performed in sequence. Variable source code has to be separated from kernel source code in the separation of concerns step. The SPL customization has to be performed next to select the target application features before integrating variable source code with kernel source code in the code weaving step. The customization file generated in the SPL customization step is required in the integration process. Target applications are compiled to produce an executable target application.

(SCAC Pattern – Continue)

Step 1: Separation of concerns between kernel and variable source code
This step involves separating kernel source code from optional and alternative source code into a variable source code file where separated source code is grouped by features. This step is similar to the separation of concerns step in the DCAC-SC pattern, but differs in the construction of the variable source code file to include necessary decisions when more than one feature is involved within an insertion point name. These decisions enable the code weaver engine to integrate only selected variable source code rather than integrating all variable source code as done in the DCAC-SC.

Dynamics
The following scenario depicts the dynamic behavior of Separation of concerns:

- Create application classes with kernel source code.
- Create a variable source code file that contains source code related to alternative and optional features.
- Add necessary decisions within insertion point names for insertions that involve more than one feature (feature interaction).
- Add insertion points to kernel source code where optional and alternative source code from the variable source code file will be inserted, based on feature selection.


(SCAC Pattern – Continue)
Kernel Source Code
Class.......()
{
$START ins1
$START ins2
$START ins3
$START ins4
}
Variable source code File
$FEATURE[A] // Optional Feature
$START ins1
// Insertion code
$END ins1
$START ins2
// Insertion code
$END ins2
$ENDFEATURE[A]
$FEATURE[X] // Alternative Feature
$START ins3
// Insertion code
$END ins3
$ENDFEATURE[X]
$FEATURE[Y] // Alternative Feature
$START ins3
// Insertion code
$END ins3
$ENDFEATURE[Y]
$FEATUREINTERACTION[C,D]
$START ins4
$IF FEATURE[C,D] //Both features selected
// Insertion code
$ELSEIF FEATURE[C] //Only feature C selected
// Insertion code
$ELSEIF FEATURE[D] //Only feature D selected
// Insertion code
$ENDIF
$END ins4
$ENDFEATUREINTERACTION[C,D]

(SCAC Pattern – Continue)

Language description:

- Kernel source code
  - $START <<insertion name>>: Used to specify insertion location in kernel source code

- Variable source code file
  - $START <<insertion name>>: Used to identify optional or alternative source code that needs to be inserted at the location specified in the kernel source code.

  - $END <<insertion name>>: Specifies the end of insertion source code.

  - FEATURE [<<feature name>>]: Groups optional or alternative source code in a feature block. Feature blocks are integrated with kernel source code during the code weaving step based on insertion names.

  - FEATUREINTERACTION[<<feature 1, feature 2, …>>]: Groups related feature source code that requires decision on which source code is to be included in the code weaving step.

  - $IF FEATURE [<<feature 1>>, <<feature 2>>, ..]: A programmatic decision point within the FEATUREINTERACTION block that is used to notify the code weaver engine whether to include the following source code block or not based on selected features in the customization file.

  - $ELSEIF FEATURE [<<feature name>>]: A programmatic *ELSEIF* point to be used in case the IF FEATURE statement is false.

  - $ENDIF: Specifies the end of the decision statements.

  - ENDFEATUREINTERACTION []: Specifies the end of feature interaction source code.

(SCAC Pattern – Continue)

Step 2: SPL Customization
This step is identical to the SPL customization step in the DCAC and DCAC-SC patterns. However, this step has to be performed before integrating variable source code with kernel source code in the code weaving step. It involves selecting desired optional and alternative features to be included in the target application. The feature selector component provides a facility to make feature selection from the feature model and run consistency checks to verify feature selections. Once features are selected, selection information will be stored in the customization file by the customization file generator. The code weaver component reads this file to integrate selected feature source code with kernel source code.

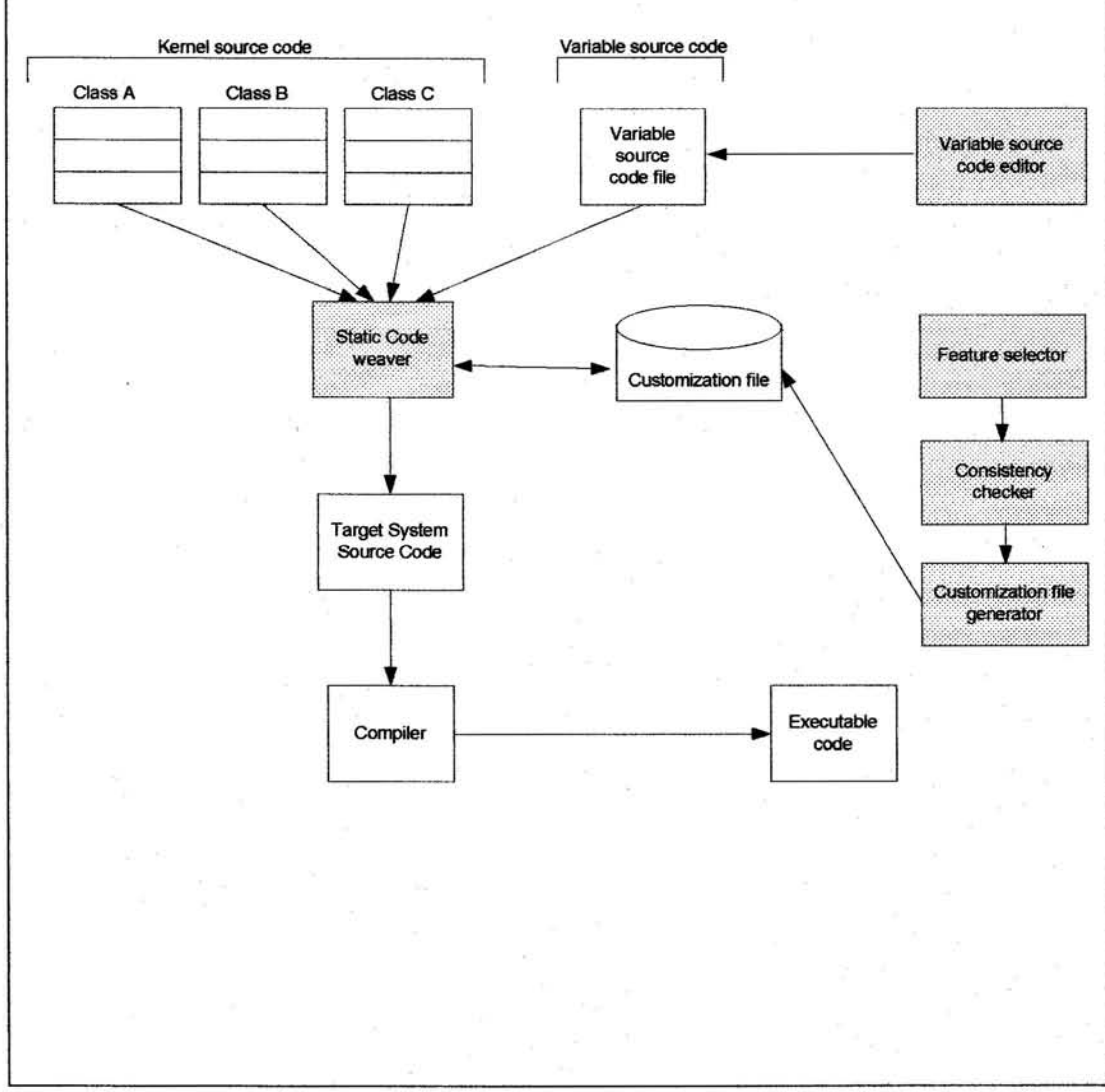
Step 3: Code weaving
This process combines kernel source code with optional and alternative source code from the created variable source code file and the customization file. This step is based on a source code integration engine, which reads the variable source code file code and inserts only selected source code that is related to selected features into the kernel source code at the specified insertion locations. This means, if an optional feature is selected, its related source code in the variable source code file will be inserted in the target system, and if one or the other alternative feature is selected, only related source code of the selected alternative feature is inserted in the target system at the location of the insertion point. Feature grouping and insertion points are the key for separation of concerns and source code integration.

(SCAC Pattern – Continue)

Dynamics

The following scenario depicts the dynamic behavior of code weaving step:

- Run the code weaver component.
- Read selected optional and alternative source code from the variable source code file and integrate it into kernel classes at the specified insertion point locations. The generated customization file is used for making decisions on which feature source code to insert.
- Compile integrated source code to generate an executable target system with only the required target system source code.



(SCAC Pattern – Continue)

The following diagram shows the complete processes of separation of concerns, feature selection, and code weaving:

Create Kernel source code in classes
Create client application classes
Kernel source code

Add Insertion points where source code from the variable source code file will be inserted based on feature selection
Add insertion points in classes

Create a variable source code file that contains source code related to alternative and optional features
Create variable source code file
Variable source code file

Select target system features and run consistency checks
Select features

Read selected features and integrate related source code from the feature file into kernel classes
Weave code
Kernel source code

Compile the integrated source code to generate an executable target system
Compile
Integrated source code

Run Executable target application
Executable code

(SCAC Pattern – Continue)

Step 4: Target application interaction

Once the interactive application is integrated and compiled, it will have the following components structure:

- User interface component
- Web service component

User interface component is responsible for accepting input from users and allowing invocation of possible service requests. It involves the sequencing of web services invocation and handling of message communication based on the customized workflow. It is also responsible for displaying results to users received from the web service component.

Web Service component is a collection of functional methods that are packaged as a single unit and published in the Internet for use by other software programs, in this case the user interface component.

Class Web service	**Collaboration**
Responsibility - Process a service request based on provided input - Returns results of processed request	- User interface

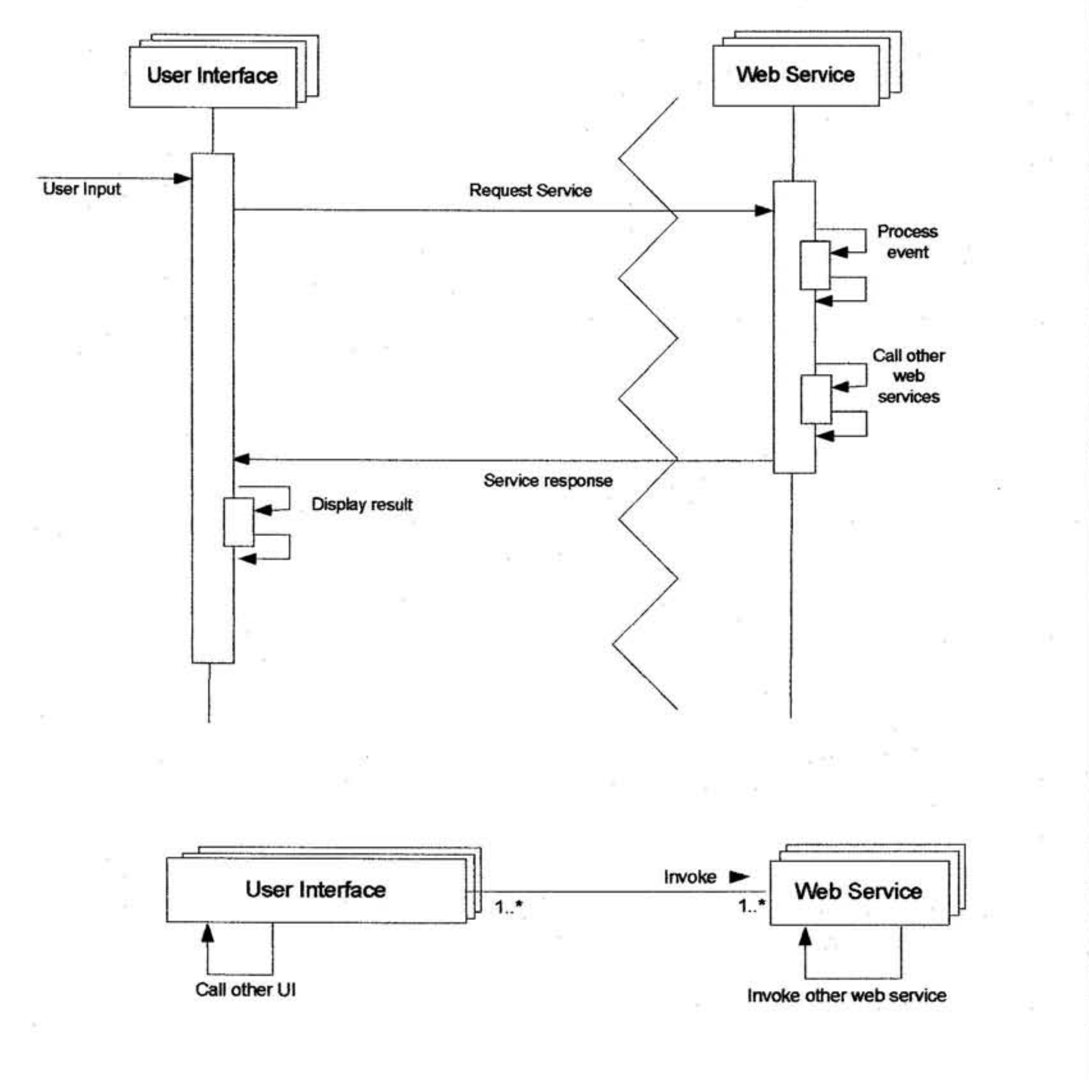
Class User interface	**Collaboration**
Responsibility - Accepts user input and service request - Invoke and pass parameters to appropriate web service(s) - Receives results from web service(s) - Display information to the user	- Web service

(SCAC Pattern – Continue)

Dynamics

The following scenario shows how service requests are processed using SCAC:

- User invokes a user interface
- User requests a service by entering input data and clicking a button
- User interface passes the service request and input data to a web service method(s).
- Web service processes request and returns results to the user interface. A web service may also request service from other web services.
- User interface displays results received from web service.



CURRICLUM VITAE

Mazen Saleh was born on June 13, 1968, in Makkah, Saudi Arabia. In 1990 he received his B.Sc. in Computer Science from Texas Southern University at Houston, Texas. He obtained a Master of Science in Computer Information Systems from American University at Washington, DC, in 2000. He joined a doctoral program at George Mason University in Fall 2001.

From 1991 to 1999, Mr. Saleh worked for the Ministry of Telecommunications in Saudi Arabia. He started as a systems analyst and was promoted to director of the Information Technology department of the Radio Frequency Division in 1995.

العنوان:	Software Product Line Engineering Based on Web Services
المؤلف الرئيسي:	Saleh, Mazen M. Aquil
مؤلفين آخرين:	Gomaa, Hassan(Super.)
التاريخ الميلادي:	2005
موقع:	فيرفاكس، فرجينيا
رقم MD:	618453
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	البرمجيات، الإنترنت، تقنية المعلومات، هندسة الحاسبات
رابط:	https://search.mandumah.com/Record/618453

© 2019 دار المنظومة. جميع الحقوق محفوظة.
هذه المادة متاحة بناء على الإتفاق الموقع مع أصحاب حقوق النشر، علما أن جميع حقوق النشر محفوظة. يمكنك تحميل أو طباعة هذه المادة للاستخدام الشخصي فقط، ويمنع النسخ أو التحويل أو النشر عبر أي وسيلة (مثل مواقع الانترنت أو البريد الالكتروني) دون تصريح خطي من أصحاب حقوق النشر أو دار المنظومة.

www.manaraa.com

Software Product Line Engineering Based on Web Services

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctoral of Philosophy at George Mason University.

By

Mazen M. Aquil Saleh

Bachelor of Science, Texas Southern University, 1990
Master of Science, American University, 2000

Director: Dr. Hassan Gomaa
Professor, Information and Software Systems Engineering

Spring Semester 2005
George Mason University
Fairfax, Virginia